



Universidad Nacional Autónoma de México
Programa de Servicio Social

“Estudio, diseño e implementación de algoritmos
paralelos usando GPU’s”

Reporte Técnico

Gpu's y lenguajes de alto nivel

Grupo de trabajo:
Diana Paola Sanjuan Aldape
Francisco Guerrero Ferrusca

Docente:
Prof. Oscar A. Esquivel Flores

Semestre 2022-II

Índice general

1. El lenguaje de programación Julia	5
1.1. ¿Qué es Julia?	5
1.2. Interactuando con Julia	5
1.2.1. El REPL	6
1.2.2. VS Code y otros IDE's para Julia	7
1.2.3. Jupyter Notebook	7
1.3. Conceptos fundamentales	8
1.3.1. Variables	8
1.3.2. Tipos de datos	9
1.3.3. Funciones y métodos	13
1.3.4. Despacho múltiple	14
1.4. Julia es rápido	15
1.4.1. Benchmarking	16
1.4.2. Comparando el rendimiento de Julia	17
1.5. Como hacer Julia mas rápido	17
1.5.1. Perfilado	17
1.5.2. Evitar los tipos abstractos	18
1.5.3. Evitar el alcance global	18
1.5.4. Preasignación de memoria	19
1.5.5. Utilizar operaciones vectoriales fusionadas	20
1.5.6. Paralelismo en Julia	20
2. Unidades de procesamiento gráfico	23
2.1. Necesidad de las GPU	23
2.2. ¿Qué son las GPU's	24
2.3. Diferencia entre la GPU y la CPU	25
2.4. Elementos que integran a la GPU	25
2.5. Arquitectura de la GPU	26
2.5.1. Arquitecturas NVIDIA	28
2.5.2. Arquitecturas AMD	30
3. CUDA-C	33
3.1. Preámbulo	33
3.2. Introducción a CUDA-C	33
3.2.1. ¿Cómo funciona CUDA?	34
3.2.2. Modelo de programación CUDA	34
3.2.3. Núcleos de CUDA	35
3.3. Hola Mundo con CUDA	35
3.3.1. Kernel	35
3.3.2. ¿Cómo es la ejecución del Kernel?	36
3.3.3. Invocaciones al Kernel	36
3.3.4. Lanzamiento de un Kernel	37
3.4. Hilos en CUDA	37
3.4.1. Paralelismo de los hilos	38
3.4.2. Ejemplo	40
3.5. Memoria en CUDA	40
3.5.1. Jerarquía de memoria en CUDA	41

3.5.2. Memoria compartida	42
3.5.3. Ejemplo memoria compartida	42
3.5.4. Memoria constante	42
3.5.5. Ejemplo memoria constante	43
3.5.6. Matriz cuadrada	43
3.5.7. Sincronización	44
3.6. Arquitectura de CUDA	44
3.6.1. ¿Con que GPU's funciona CUDA?	45
3.7. Aplicaciones	46
3.7.1. Imágenes Digitales	46
3.7.2. Imágenes en color	47
3.7.3. Imágenes en mapa de bits	48
3.7.4. OpenGL	48
4. CUDA con Julia	51
4.1. Introducción a CUDA.jl	51
4.2. Hola Mundo en CUDA.jl	52
4.2.1. Solución de problemas	52
4.3. Programación en CUDA.jl	53
4.3.1. CuArraytipo	53
4.3.2. Programación del núcleo con @cuda	54
4.3.3. Envolturas de la API de CUDA	54
4.4. Abstracciones de orden superior	54
4.4.1. Construcción e inicialización	55
4.4.2. Álgebra lineal	55
4.4.3. Matrices dispersas	56
4.5. Gestión de la memoria	57
4.5.1. Preservación de tipo	58
4.5.2. Grupo de memoria	58
4.6. Hilos	59
4.6.1. Subprocesos múltiples	60
4.7. Ejecución de un Kernel	61
4.8. Curiosidades de CUDA.jl	61

Capítulo 1

El lenguaje de programación Julia

1.1. ¿Qué es Julia?

Julia fue lanzado en 2012 por Alan Edelman, Stefan Karpinski, Jeff Bezanson y Viral Shah. Se trata de un lenguaje de programación dinámico de código abierto, de alto nivel y de alto rendimiento.

El lenguaje de programación Julia ha sido diseñado para disponer de las ventajas de un lenguaje dinámico con el rendimiento de un lenguaje compilado. Esto se consigue en parte gracias a la utilización de un compilador JIT (just-in-time) basado en LLVM (Low Level Virtual Machine) que permite generar código de máquina completamente nativo.

Julia fue desarrollado en el MIT, tiene una sintaxis tan amigable como Python y un rendimiento tan competitivo como C. Dentro las características mas importantes de Julia se encuentran:

- Está desarrollado como un lenguaje de programación de alto rendimiento.
- Usa envío múltiple (“multiple dispatch” en inglés), que le permite al programador elegir entre diferentes patrones de programación de acuerdo a la aplicación.
- Es un lenguaje de tipo dinámico que se puede usar fácilmente de forma interactiva.
- Tiene una sintaxis de alto nivel que es fácil de aprender.
- Es un lenguaje de programación con tipos opcionales, cuyos tipos de datos (definidos por el usuario) hacen que el código sea claro y robusto.
- Cuenta una biblioteca estándar extendida, además, están disponibles numerosos paquetes de terceros.

Julia es muy popular en el campo del Machine Learning gracias a sus paquetes de Machine Learning de alta velocidad y a una sintaxis altamente expresiva y útil para el desarrollo del aprendizaje automático.

Este lenguaje de programación dinámico y de alto nivel está diseñado para abordar las necesidades del análisis numérico de alto rendimiento y la ciencia computacional. La biblioteca base escrita en Julia contiene:

- Las mejores bibliotecas open source C y Fortran para el álgebra lineal.
- La generación de números aleatorios.
- El procesamiento de señales.
- El procesamiento de cadenas.

Julia incluye Flux, un framework o librería de aprendizaje automático muy utilizado también en Inteligencia Artificial.

Flux proporciona una interfaz altamente intuitiva, es uno de los frameworks más flexibles que existen, ya que se puede integrar fácilmente con otras librerías o incluso trabajar con él en varios kernels a la vez.

1.2. Interactuando con Julia

Es posible leer y ejecutar código en Julia de dos formas: mediante el Julia REPL y por medio de un IDE (Entorno de desarrollo integrado), los mas comunes son: Jupyter y Visual Studio Code.

1.2.1. El REPL

La forma más rápida de operar con Julia es trabajando de forma interactiva, en lo que se llama el "REPL", por las siglas de Read-Eval-Print-Loop ("bucle leer-evaluar-imprimir"). Este proceso se realiza desde una terminal de comandos –a la que por extensión también se le da el nombre de REPL–, que es lo que se presenta al usuario al lanzar el programa (Figura 1.1.).

```

julia> 

```

The screenshot shows the Julia REPL interface. It features a decorative logo on the left consisting of various colored brackets and parentheses. To the right of the logo, the following text is displayed:

- Documentation: <https://docs.julialang.org>
- Type "?" for help, "]?" for Pkg help.
- Version 1.7.1 (2021-12-22)
- Official <https://julialang.org/> release

A green prompt "julia>" followed by a blank line is at the bottom of the screen.

Figura 1.1: REPL/consola de Julia

Instalación del REPL en Linux

- Verificar la arquitectura de la computadora utilizando el comando `uname -i`:

```
pao@matrix:~$ uname -i
```

Si la maquina es de 64 bits el resultado será `x86_64`, si es de 32 bit el resultado sera `i686`.

```
pao@matrix:~$ uname -i
x86_64
```

Donde `pao` es el nombre de usuario y `matrix` es el nombre de la máquina. Cuando corresponda se deberán reemplazar estos nombres por los de su usuario y computadora respectivamente.

- Abrir la pagina oficial de Julia: <https://julialang.org/downloads/>
- Buscar la versión estable actual y descargar el archivo comprimido **Generic Linux on x86 [help]** correspondiente a la arquitectura de la máquina previamente obtenida.
- Mover el archivo preferentemente al `home` o a un directorio específico, por ejemplo:

```
\home\pao\
```

- Descomprimir y extraer el archivo comprimido utilizando el comando `tar`:

```
tar -xvzf julia-1.7.1-linux-x86_64.tar.gz
```

Los archivos se extraen en un directorio llamado `Julia-1.7.1`. Nota:

Los valores 1.7.1 corresponden a la versión actual de Julia, por lo que se tendrán que modificar dependiendo de la versión descargada.

- Agregar Julia a la variable PATH de Linux para que Julia corra desde cualquier directorio, para esto hay que agregar el directorio bin que contiene el comando ejecutable `julia` a la variable PATH de la siguiente manera: Editar el archivo `bash_profile` (si existe), o sino el archivo `.bashrc`. Abrir uno de esos archivos en en cualquier editor de texto y agregar al principio de todo la siguiente línea:

```
export PATH="$PATH:<ruta>/Julia-1.7.1/bin"
```

Donde ruta es la ubicación de la carpeta Julia-1.7.1 que en este caso es \home\pao\ de modo que quedaría de la siguiente manera:

```
export PATH="$PATH:/home/pao/Julia-1.7.1/bin"
```

VII. Abrir terminal e ingresar el comando **julia**

```
pao@matrix:~$ julia
```

Se debe visualizar lo que se muestra en la Figura 1.1.

1.2.2. VS Code y otros IDE's para Julia



Figura 1.2: Entorno de VS Code

Las interfaces basadas en una consola de comandos como el REPL resultan poco "amigables"; y por otro lado, para ejecutar rutinas más complejas, y siempre que se quiera obtener resultados reproducibles, es recomendable escribir las instrucciones en un archivo de código (script).

Para combinar ambas tareas de forma eficiente en una sola interfaz lo habitual es usar los llamados "entornos de desarrollo integrados" (IDE), que juntan en una misma interfaz una consola de comandos, un editor de código, y a menudo otras utilidades como pueden ser visores de variables, tablas y gráficas, paneles de documentación, herramientas de depuración, etc.

Julia cuenta con plug-ins para crear IDEs sobre editores de código avanzados, como VS Code, Atom, Emacs, Sublime Text, entre otros.

El IDE más completo y popular es el basado en VS Code. Para trabajar en ese entorno, además de Julia, hay que instalar VS Code, y activar su extensión para Julia (Figura 1.2).

1.2.3. Jupyter Notebook

Los cuadernos de código "(notebooks)", combinan en un mismo documento el código a ejecutar, las salidas (tablas, gráficos y otros resultados), y también texto libre con explicaciones y comentarios. La aplicación

más conocida para crear y visualizar ese tipo de notebooks es Jupyter, que admite varios lenguajes de programación, incluyendo Julia. Jupyter funciona sobre Python, y para hacer notebooks de Julia hay que instalar el paquete IJulia, que también incluye una instalación básica de Python.

Para instalar IJulia es necesario iniciar el REPL de Julia e ingresar los siguientes comandos:

```
julia> using Pkg
julia> Pkg.add("IJulia")
```

El proceso puede tardar unos minutos. Cuando termine se mostrara lo siguiente (Figura 1.3):

```
D:\Programas\VS Code\Julia\julia-1.7.3\bin\julia.exe
[ea8e919c] + SHA
[9e88b42a] + Serialization
[6462fe0b] + Sockets
[fa267f1f] + TOML
[a4e569a6] + Tar
[8dfed614] + Test
[cf7118a7] + UIDs
[4ec0a83e] + Unicode
[deac9b47] + LibCURL_jll
[29816b5a] + LibSSH2_jll
[c8ffd9c3] + MbedTLS_jll
[14a3606d] + MozillaCACerts_jll
[83775a58] + Zlib_jll
[8e850ede] + nghttp2_jll
[3f19e933] + p7zip_jll
Building Conda → `C:\Users\B-DRIVE IT\.julia\scratchspaces\44cf95a-1eb2-52ea-b672-e2afdf69b78f\6e47d11ea2776bc5627
421d59cdcc1296c058071\build.log`
Building IJulia → `C:\Users\B-DRIVE IT\.julia\scratchspaces\44cf95a-1eb2-52ea-b672-e2afdf69b78f\98ab633acb0fe071b67
1f6c1785c46cd70bb86bd\build.log`
Precompiling project...
11 dependencies successfully precompiled in 7 seconds (4 already precompiled)

julia>
```

Figura 1.3: Instalación completa

Finalmente utilizando los comandos:

```
julia> using IJulia
julia> jupyterlab()
```

Se debe visualizará una ventana de Jupyter como la que se muestra en la Figura 1.3.

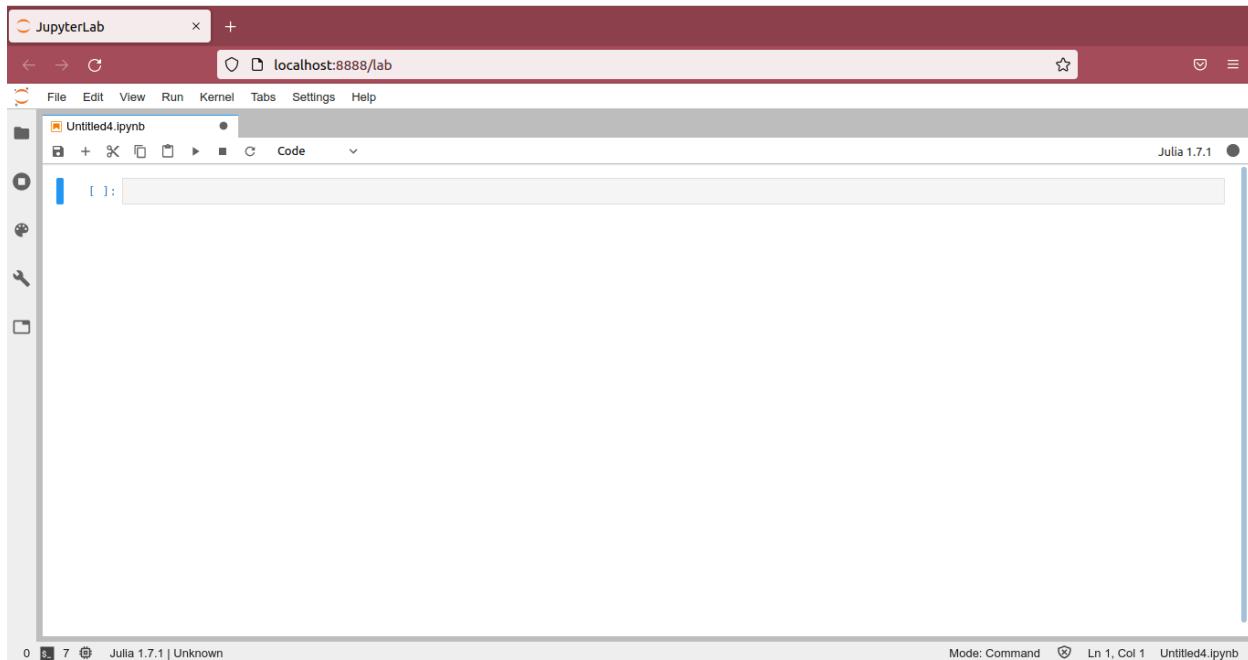
1.3. Conceptos fundamentales

1.3.1. Variables

Una variable, en Julia, es un nombre asociado a un valor. Es útil cuando se requiere almacenar un valor para usarlo posteriormente. Julia proporciona un sistema extremadamente flexible para nombrar variables. Los nombres de las variables deben comenzar con una letra (AZ o az), un guión bajo o Unicode mayor que 00A0; en particular, las categorías de caracteres Unicode Lu / Ll / Lt / Lm / Lo / Nl (letras), Sc / So (moneda y otros símbolos) y algunos otros caracteres similares a letras son permitido. Los caracteres siguientes también pueden incluir ! y dígitos (0-9), así como otros puntos de código Unicode: diacríticos y otros signos de modificación, algunos conectores de puntuación, primos y algunos otros caracteres.

Los operadores como + también son identificadores válidos, pero se analizan especialmente. En algunos contextos, los operadores se pueden utilizar como variables; por ejemplo (+) se refiere a la función de suma. Ejemplos:

```
julia> x = 2
2
```

**Figura 1.4:** Jupyter Notebook

```
julia> y = x + 10
12

julia> valor = 20.3
20.3

julia> saludo_inicial = "Hola!"
"Hola!"

julia> μ = 0.0001
0.0001
```

1.3.2. Tipos de datos

Los tipos en Julia son básicamente los elementos de datos de varios tamaños y funcionalidades diferentes. Cada valor (no variable) en un programa está destinado a ser de algún tipo. El sistema de tipos de Julia se divide básicamente en dos categorías, según el momento de su definición. Estas son:

- Sistema de tipo estático: en un sistema de tipo estático, cada expresión de programa ejecutable se define con un tipo antes de su ejecución. Definir el tipo es factible para una mayor velocidad de ejecución del código.
- Sistema de tipo dinámico: en un sistema de tipos dinámicos, el compilador tiene que decidir el tipo de valores en el momento de la compilación del código. Esto hará que el código tarde más en ejecutarse.

Asignación de tipos en Julia

El operador para indicar el tipo de dato en Julia es ::. Generalmente se indica el nombre de una variable, el operador ::, seguido del tipo de dato. El operador lo que hace es comparar el dato ingresado con el tipo, en dado caso de no coincidir manda un error. Así como se muestra a continuación.

```
Julia> 25::Float64
TypeError: in typeassert, expected Float64, got Int64
```

Los tipos de datos se pueden dividir en diferentes tipos que se denominan subtipos (por ejemplo String, bool, etc.).

Tipos abstractos

Los tipos abstractos no pueden ser instanciados y sirven sólo como nodos en el grafo de tipos, describiendo así conjuntos de tipos concretos relacionados, aquellos tipos concretos que son sus descendientes.

Los tipos concretos no pueden tener subtipos.

Los tipos abstractos aunque no tengan instanciación, forman la jerarquía conceptual que hace que el sistema de tipos de Julia sea algo más que una colección de implementaciones de objetos. Es decir, están destinados a actuar únicamente como supertipo de otros tipos. Permiten agrupar a diferentes subtipos, pero pueden usarse como una anotación de tipo.

Signed, Real y Nomber son ejemplos de Tipos Abstractos.

Las sintaxis generales para declarar un tipo abstracto son:

```
abstract type <name> end
abstract type <name> <: <supertype> end
```

Tipos primitivos

Como se mencionó anteriormente los tipos concretos son aquellos que pueden tener instancias. En Julia los tipos primitivos son los tipos concretos cuyo valor está en forma de bits. Los datos enteros y flotantes son un ejemplo de tipos primitivos. Julia tiene un conjunto predefinido de tipos primitivos estándar, pero también permite declarar sus propios tipos primitivos.

Las sintaxis generales para declarar un tipo primitivo son:

```
primitive type <name> <bits> end
primitive type <name> <: <supertype> <bits> end
```

Jerarquía de tipos

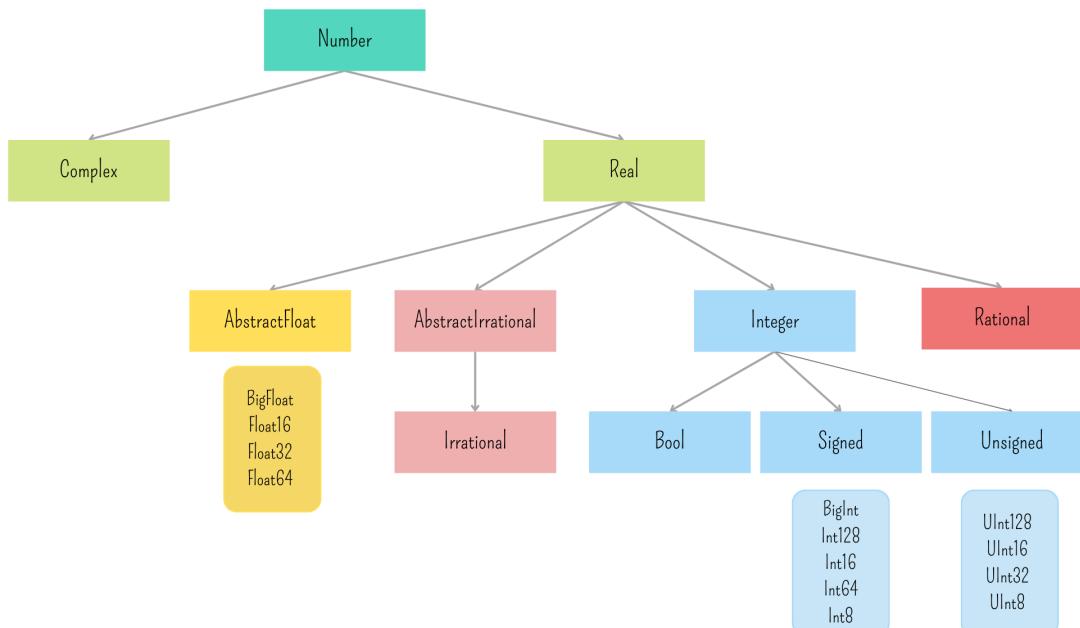


Figura 1.5: Árbol jerárquico de *Number*

En Julia los tipos de datos son objetos, por eso unos heredan de otros. Por ejemplo, el tipo Int64 hereda de Signed, este a su vez hereda de Real que hereda de Number el cual hereda del tipo base Any.

En la figura 1.5 se muestra el diagrama de la jerarquía del tipo Number.

Tipos compuestos

Los tipos de datos compuestos (*composite types*) también conocidos como tipos de datos definidos por el usuario, son una colección de campos con nombre que se puede tratar individualmente como valores únicos de tipos específicos. Es decir, se pueden crear tipos de datos personalizados usando los tipos de datos del lenguaje.

Los tipos compuestos se pueden declarar con el uso de la palabra clave **struct**:

```
1 struct Datos
2     nombre::String
3     edad::Int64
4     peso
5 end
```

Los campos que no se especifican con ningún tipo se establecen con el tipo predeterminado **Any**. El objeto de este tipo compuesto se puede crear con el uso de constructores.

```
6 obj1 = Datos("Luis", 20, 78.2)
```

```
Datos("Luis", 20, 78.2)
```

Los tipos compuestos creados con el uso de la palabra clave **struct** son inmutables, lo que significa que no se pueden modificar después de su creación.

Como se observa a continuación:

TIPOS COMPUESTOS MUTABLES

Julia permite crear tipos compuestos de tipos mutables, lo que significa que el valor de su campo se puede modificar incluso después de su creación. Esto se hace mediante el uso de las palabras clave **mutable struct**:

```
1 mutable struct Producto
2     Nombre::String
3     Precio::Float64
4     Descuento::Bool
5 end
```

Estos tres tipos de datos están estrechamente relacionados entre sí y sus propiedades son similares:

- Todos se declaran explícitamente.
- Siempre se definen con sus nombres.
- Los supertipos de los tres tipos se declaran explícitamente.
- Pueden contener algunos parámetros.

Estos tipos son todas instancias de un solo tipo llamado **DataType**. Un **DataType** puede ser de tipo abstracto o concreto según las propiedades. Cada valor concreto es una instancia de algún **DataType**. Si es primitivo, entonces tiene un tamaño distinto de cero sin un nombre de campo especificado y para el tipo de datos compuesto debe contener el nombre del campo y puede estar vacío.

Ejemplo:

```
julia> typeof(Int)
Datatype
```

```
julia> typeof(Bool)
Datatype
```

```
julia> typeof(Float)
Datatype
```

Tipos parámetro

Julia permite pasar parámetros a sus tipos, lo que da como resultado la generación de un rango completamente nuevo de tipos, uno para cada combinación de valor de parámetro. Todos los tipos declarados denominados **DataType** en general, se pueden parametrizar. Esta declaración de tipo parametrizado sigue la misma sintaxis para cada una de las posibilidades. Hay tres tipos de tipos paramétricos:

- Tipos abstractos paramétricos Los tipos abstractos paramétricos se declaran de forma similar a como se declararon los tipos abstractos. En un tipo abstracto paramétrico, se usa el tipo abstracto de prefijo, los parámetros se escriben después del nombre del tipo encerrado entre llaves. El parámetro aquí es de tipo T, donde T puede ser cualquier tipo.

Sintaxis:

```
abstract type {T} end
```

- Tipos primitivos paramétricos Los tipos primitivos paramétricos se declaran de forma similar a como se declararon los tipos primitivos. En un tipo primitivo paramétrico se usa el tipo primitivo de prefijo, los parámetros se escriben después del nombre del tipo encerrado entre llaves. El parámetro aquí es de tipo T, donde T puede ser cualquier tipo.

Sintaxis:

```
primitive type {T} 32 end #Para 32 bits
primitive type {T} 64 end #Para 64 bits
```

- Tipos compuestos paramétricos En un tipo compuesto paramétrico, los parámetros se escriben después del nombre del tipo entre llaves. El parámetro aquí es de tipo general T, donde T puede ser cualquier tipo.

Sintaxis:

```
1 struct Punto{T}
2     x::T
3     y::T
4 end
5
```

Se ingresan valores enteros a la estructura:

```
6 Punto(5, 3)
```

```
Punto{Int64}(5, 3)
```

Se ingresan valores flotantes a la estructura:

```
7 Punto(3.7, 4.2)
```

```
Punto{Float64}(3.7, 4.2)
```

Se ingresan cadenas a la estructura:

```
8 Punto("Oaxaca", "Puebla")
```

```
Punto{String}("Oaxaca", "Puebla")
```

T se puede reemplazar con cualquier tipo concreto como Int64, Float64, AbstractString, etc. Cada uno de estos se convertirá en un tipo concreto utilizable.

También es posible limitar los tipos de datos indicando que T solo pertenece a un subconjunto de los tipos. En este caso, se puede limitar para que los datos sean únicamente de tipo numérico.

```
1 struct Punto{T <: Number}
2     x::T
3     y::T
4 end
```

Se ingresan valores enteros a la estructura:

```
5 Punto(5, 3)
```

```
6 Punto{Int64}(5, 3)
```

Se ingresan valores flotantes a la estructura:

```
6 Punto(3.7, 4.2)
```

```
Punto{Float64}(3.7, 4.2)
```

Se ingresan cadenas a la estructura:

```
7 Punto("Oaxaca", "Puebla")
```

```
MethodError: no method matching Punto(::String, ::String)
```

Se genera un error dado que se restringió a números únicamente,

1.3.3. Funciones y métodos

Una función en programación es una secuencia de sentencias que ejecuta una operación deseada, puede o no devolver valores.

Definición de funciones en Julia:

Para definir una función se debe especificar su nombre y la secuencia de sentencias de la siguiente manera:

```
1 function suma(x,y)
2     println(x + y)
3 end
```

```
suma (generic function with 1 method)
```

Una vez definida la función, se "llama" de la siguiente manera:

```
4 suma(5, 3)
```

Julia admite que se indique el tipo de dato para una función. Lo que permite sobrecargar las funciones creando una para cada tipo de dato diferente.

```
1 function factorial(n::Int64)
2     if n == 0
3         return 1;
4     else
5         return n * fact(n-1)
6     end
7 end
```

```
factorial (generic function with 1 method)
```

Llamada a la función:

```
8 factorial(3)
```

En Julia las funciones pueden ser definidas por partes proporcionando comportamientos específicos para ciertas combinaciones de tipos de argumentos. La definición de un posible comportamiento de una función se denomina método.

La elección de qué método se ejecuta cuando se aplica una función se llama despacho. El uso de todos los argumentos de una función para elegir qué método debe ser invocado, en lugar de sólo el primero, se conoce como **despacho múltiple**

.

1.3.4. Despacho múltiple

Julia es un lenguaje de programación en el cuál los tipos de variables son claves para hacer los procesos eficientes, la filosofía del despacho múltiple descansa sobre esta idea. Despacho múltiple significa que las funciones despacharán distintos procesos (llamados métodos en Julia) según el tipo de variables que hayan sido provistos como inputs. La clave entonces es que una vez que se compila un procedimiento, el método particular asociado al tipo de variables será usado siempre con esa variable (note que la clave entonces es que el tipo no cambie, esto es exista estabilidad de tipos). Esto es lo que hace que Julia sea bastante eficiente. Por supuesto, si un programa tiene inestabilidad de tipos, los métodos irán cambiando y será necesario compilar nuevamente los procesos (hecho que ocurre instantáneamente por el JIT). Esto puede hacer que Julia sea tan lento e inefficiente como otros lenguajes de alto nivel. Por esta razón Julia permite declarar explícitamente y de forma simple los tipos de variables.

Si el programador no provee los tipos de variables explícitamente, Julia “adivina” el tipo y aplica el método correspondiente.

Un ejemplo es la siguiente función, en la que no se define explícitamente el tipo de su input:

```
1 function multiplica(x,y)
2     return x*y
3 end
```

```
multiplica (generic function with 1 method)
```

Al aplicar la función a una variable `Float64`, Julia aplica el método correspondiente a la multiplicación de variables `Float64`:

```
multiplica(2.0,3.0)
```

```
out: 6.0
```

Si se aplica la función a una variable `String`, Julia aplica el método correspondiente que en este caso es concatenar variables `String`:

```
multiplica("hola ","Pedro")
```

```
out: "hola Pedro"
```

Esto ocurre porque Julia sabe que hacer dependiendo del tipo de x.

Para ejemplificar un poco esto se creó una función llamada `fun_tipo` en donde se declara explícitamente el tipo del input. Si el input es un `Int64`, la función imprimirá en pantalla que es de tipo `Int64`:

```
1 function fun_tipo(a::Int64)
2     println("$a es de tipo Int")
3 end
```

```
fun_tipo (generic function with 1 method)
```

Esto genera una función genérica llamada `fun_tipo` con un método.

A continuación, se crea la misma función pero en este caso se le dice que si el input es un `Float64` la función imprima en pantalla que es de tipo `Float64`:

```
4 function fun_tipo(a::Float64)
5     println("$a es de tipo Float")
6 end
```

```
fun_tipo (generic function with 2 methods)
```

Aquí, Julia indica que se creó `fun_tipo` con 2 métodos.

Finalmente, se define de nuevo la función pero ahora con `String`. En este caso, la función imprimirá en pantalla que se tiene un `String`:

```
7 function fun_tipo(a::String)
8     println("$a es de tipo String")
9 end
```

```
fun_tipo (generic function with 3 methods)
```

Lo que genera una función genérica llamada *fun_tipo* con 3 métodos.

Para verificar los métodos de la función se emplea la función:

```
10 methods(fun_tipo)
```

```
3 methods for generic function "fun_tipo":  
[1] fun_tipo(a::Int64) in Main at REPL[4]:1  
[2] fun_tipo(a::Float64) in Main at REPL[5]:1  
[3] fun_tipo(a::String) in Main at REPL[6]:1
```

Si al llamar una función no se especifica el tipo de las variables Julia adivinará el método.

Al ejecutar la función con distintos tipos de datos, los resultados son los siguientes:

```
fun_tipo(3)  
2 es de tipo Int64
```

```
fun_tipo(2.5)  
2.5 es de tipo Float64
```

```
fun_tipo("Hello world")  
Hello world es de tipo String
```

Pero al pasar un argumento de tipo Bool se obtiene un error, porque aun no se ha definido con este tipo de variable:

```
fun_tipo(true)  
  
MethodError: no method matching fun_tipo(::Bool)  
Closest candidates are:  
  fun_tipo(!Matched::String) at In[7]:2  
  fun_tipo(!Matched::Float64) at In[6]:2  
  fun_tipo(!Matched::Int64) at In[5]:2
```

Stacktrace:

```
[1] top-level scope at In[12]:1
```

1.4. Julia es rápido

Una de las principales razones del porque Julia es uno de los lenguajes mas rápidos comparado con otros lenguajes de computación científica (Python, R, Matlab) es que cuenta con un compilador JIT.

Un compilador normal es un programa que traduce todo código escrito en un lenguaje de programación (código fuente) en un lenguaje que la máquina entienda, es decir, convierte el código a binario y genera un ejecutable. Mientras que el compilador JIT lo que hace es optimizar esta labor compilando tan solo el código de cada función cuando es necesario.

De esta manera cuando se va a ejecutar un programa, el compilador JIT solo compilará las funciones que se utilizarán en ese momento, guardando el resultado en una caché. A medida que se utiliza el programa, cuando se encuentra con una nueva función que aún no se ha compilado, esta se compila de nuevo. Pero cuando encuentra una función que ya se ha utilizado, en lugar de compilarla de nuevo se busca en la caché, ahorrando una importante cantidad de tiempo.

Otra de las razones que no solo optimizan su velocidad, además mejoran su rendimiento es su capacidad de paralelizar el código. En Julia para ejecutar un script solamente se tiene que indicar el número de núcleos en la línea de comandos a la hora de llamar a este. Además, es posible enviar tareas a diferentes hilos directamente desde el código. Finalmente, los bucles de Julia se puede ejecutar el paralelo directamente no es necesario indicarlo como en otros lenguajes. Se explicara mas a detalle el paralelismo en secciones posteriores.

1.4.1. Benchmarking

Una prueba de rendimiento o comparativa (en inglés benchmark) es una técnica utilizada para medir el rendimiento de un sistema o uno de sus componentes. Un Benchmark también evalúa la estabilidad del sistema, ya que al exprimir sus componentes está comprobando si todos ellos funcionan perfectamente cuando están en condiciones extremas de exigencias. Otra utilidad del Benchmark es la de realizar rankings de rendimiento. En este caso lo que se busca es el rendimiento máximo de algún benchmark.

Para realizar estas pruebas de rendimiento Julia cuenta con un paquete llamado Benchmark Tools.

Para agregar el paquete Benchmark Tools se debe ingresar en el REPL de Juilia:

```
julia> import Pkg; Pkg.add("BenchmarkTools")
```

Una vez que termine la descarga ya se pueden usar las herramientas de BenchmarkTools mediante el siguiente comando:

```
julia> using BenchmarkTools
```

Las macros mas utilizadas son @benchmark y @btime. Con @benchmark es posible realizar pruebas rápidas de rendimiento, esta macro proporciona no solo estadísticas simples, como el mínimo, la mediana y el máximo de todas las muestras que ha tomado, sino que también puede acceder a todos los datos de la muestra. Es decir, si @benchmark ejecuto la función 2000 veces correctamente, podrá acceder a esas 2000 muestras. También muestra un histograma con la distribución de los datos. Mientras que la macro @btime imprime el

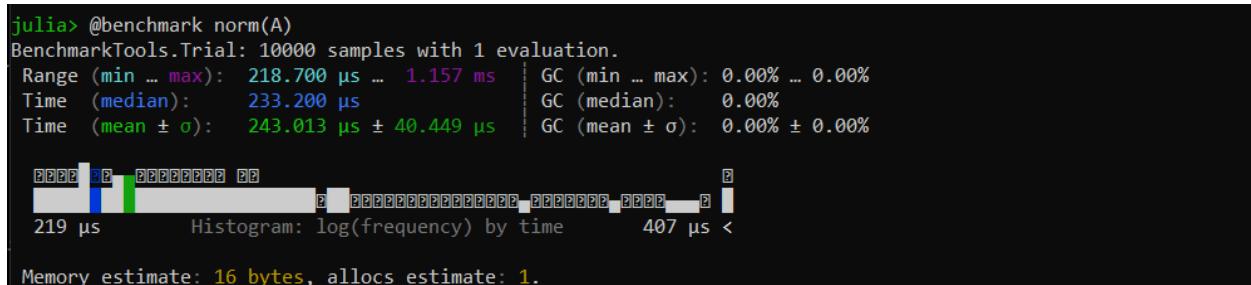


Figura 1.6: Histograma recopilado por la macro @benchmark

tiempo mínimo y la asignación de memoria antes de devolver el valor de la expresión.

```
julia> @btime norm(A)
  213.500 μs (1 allocation: 16 bytes)
576.7331602372549
```

La macro @belapsed devuelve el tiempo mínimo en segundos.

```
julia> @belapsed norm(A)
0.0002137
```

A continuación se muestran otras macros con las que cuenta el paquete Benchmark Tools así como su funcionalidad:

- **@which:** Evalúa los argumentos de la llamada a la función, determina sus tipos y llama a la función `which` sobre la expresión resultante.
- **@show:** Muestra una expresión y un resultado, devolviendo el resultado.
- **@simd:** Activa las optimizaciones SIMD. (Las optimizaciones SIMD no están activadas por defecto, porque sólo aceleran tipos de código muy particulares, y activarlas en todas partes ralentizaría demasiado el compilador).
- **@bprofile:** Crea perfiles de código.

1.4.2. Comparando el rendimiento de Julia

Python al igual que Julia es un lenguaje de programación de alto nivel muy utilizado en computación científica, pero en contrario a Julia, Python es un lenguaje interpretado, lo que lo convierte en un lenguaje con mayor portabilidad y multiplataforma.

Cabe mencionar que el interprete a diferencia del compilador ejecuta directamente las instrucciones escritas en el lenguaje de programación dado. Por esta razón los lenguajes compilados suelen ser mas rápidos que los lenguajes interpretados.

C es un lenguaje de programación de nivel medio ((beneficiándose de las ventajas de la programación de alto y bajo nivel).

Para comparar el rendimiento de Julia con respecto a Python y C se resolverá el mismo problema: La sucesión de Fibonacci.

Función en Julia

Para realizar las pruebas de rendimiento en Julia se usará la macro `@btime`, del paquete de Benchmark-Tool, esta macro devuelve el tiempo de ejecución de la función así como la asignación de memoria utilizada.

```

1 function fibonacci(n)
2     if n < 2
3         return n
4     else
5         return fibonacci(n-1) + fibonacci(n-2)
6     end
7 end

julia> @time fibonacci(20)
 32.727 μs (0 allocations: 0 bytes)
6765

```

Función en Python

```

1 def fibonacci(n):
2     if n < 2:
3         return n
4     else:
5         return fibonacci(n-1) + fibonacci(n-2)

>>> 0.0021779537200927734

```

Función en C

```

1 int fibonacci(int n)
2 {
3     if (n < 2)
4         return n;
5     else
6         return fibonacci(n-1) + fibonacci(n-2);
7 }

```

Resultados

1.5. Como hacer Julia mas rápido

1.5.1. Perfilado

El perfilado supone la recopilación de las características de un programa durante su ejecución. Al realizar el perfilado se mide el tiempo de ejecución y el número de llamadas de funciones y líneas individuales en el

código del programa.

Pasos:

1. Iniciar Julia en la terminal con el comando `julia --track-allocation=<config(user, none, all)>`
2. Escribir `using Profile`
3. Incluir los archivos necesarios para perfilar el código
4. Ejecutar el código para verificar que funciona
5. Limpiar los datos (el buffer) `Profile.clear_malloc_data()`
6. Usar la macro `@Profile miFuncion()`

Al realizar el perfilado se generará un archivo con extensión `.prof` en el cual tendrá en cada línea a la izquierda el número de bytes asignados.

Conociendo esta información es posible realizar las modificaciones necesarias para optimizar el código.

1.5.2. Evitar los tipos abstractos

Los tipos de datos abstractos como se explico en la sección 1.4.2.2. sirven como nodos dentro del árbol de tipos, mientras que los tipos concretos ya que no pueden tener subtipos son los nodos finales.

A continuación se exemplificara con la función `xxxx` el como afecta en el rendimiento del programa utilizar tipos de datos abstractos.

Salida

Como se observa la cantidad de memoria al usar el tipo concreto `Int64` es mucho menor que al usar el tipo abstracto `Any`, ya que al no declarar tipo de dato, Julia lo considera como cualquiera: `Any`. Por lo anterior, también se ahorrara tiempo en la ejecución.

1.5.3. Evitar el alcance global

El alcance es una propiedad de las variables, se refiere a su visibilidad (aquella región del programa donde la variable puede utilizarse). Los distintos tipos de variables tienen distintas reglas de alcance.

Existen dos tipos de alcance: global y local.

Un alcance global (global scope) significa que alcanza a todo el programa. Por ejemplo, una variable global puede ser vistas desde cualquier parte del programa y por tanto puede ser usadas en cualquier parte.

El alcance local son todas las variables, objetos o cosas instanciadas en funciones. Por ejemplo, una variable local (declarada en una función) solo podrá ser utilizada en la función, no se podra hacer referencia fuera de dicha sección.

Para un mejor rendimiento en Julia es importante poner el código en funciones.

```

1 a = rand(10^7)
2 function sum_global(A)
3     s = 0.0
4     for a in A
5         s += a
6     end
7     return s
8 end
9
10 function sum_local()
11     s = 0.0
12     A = rand(10^7)
13     for a in A
14         s += a
15     end
16     return s
17 end

```

```
julia> @time sum_global(a)
0.029848 seconds (22.48 k allocations: 858.211 KiB)
4.999945513111102e6

julia> @time sum_local()
0.089707 seconds (37.51 k allocations: 77.844 MiB, 6.50% gc time)
4.998575837021112e6
```

Se observa una diferencia importante en la cantidad de memoria y de asignaciones entre las dos funciones.

Las variables locales hacen que los programas de computadora sean más fáciles de depurar y mantener. Los programadores pueden determinar el punto exacto en el que un programa modifica el valor de una variable local, mientras que las variables globales pueden ser modificadas en cualquier lugar dentro del código fuente. Las variables locales son automáticas por defecto, es decir, sólo existen mientras se ejecuta la función. Cuando se invoca la función se crean estas variables en la pila y se destruyen cuando la función termina. La única excepción la constituyen las variables locales declaradas como estáticas (`static`). En este caso, la variable mantiene su valor entre cada dos llamadas a la función aún cuando su visibilidad sigue siendo local a la función.

Si se deben usar variables globales se recomienda usar `const`. Al utilizar `const` se restringe la variable a un tipo específico, pero el valor puede cambiar.

1.5.4. Preasignación de memoria

Cuando una función devuelve un Array o algún otro tipo complejo, debe asignar memoria. Generalmente, la asignación y la recolección de basura son cuellos de botella sustanciales. Es posible evitar la asignación de memoria en cada llamada a la función mediante la preasignación de la salida. Como se muestra en el ejemplo, mejora tanto en tiempo como en asignación de memoria al usar la función `loopinc_preasignacion`

```
1 function xinc(x)
2     return [x, x+1, x+2]
3 end
4
5 function loopinc()
6     y = 0
7     for i = 1:10^7
8         ret = xinc(i)
9         y += ret[2]
10    end
11    return y
12 end
13
14 function xinc!(ret::AbstractVector{T}, x::T) where T
15     ret[1] = x
16     ret[2] = x+1
17     ret[3] = x+2
18     nothing
19 end
20
21 function loopinc_preasignacion()
22     ret = Vector{Int}(undef, 3)
23     y = 0
24     for i = 1:10^7
25         xinc!(ret, i)
26         y += ret[2]
27     end
28     return y
29 end
```

```
julia> @btime loopinc()
655.597 ms (39998470 allocations: 1.12 GiB)
-1994260032
```

```
julia> @btime loopinc_preasignacion()
7.600 ms (1 allocation: 96 bytes)
-1994260032
```

1.5.5. Utilizar operaciones vectoriales fusionadas

Julia cuenta con una sintaxis de puntos que convierte cualquier función escalar en una llamada de función "vectorizada" cualquier operador en un operador "vectorizado", con la propiedad especial de que las "llamadas de puntos" se fusionan : se combinan a nivel de sintaxis en un solo bucle, sin asignar matrices temporales.

```
1 f(x) = 3x.^2 + 4x + 7x.^3;
2 x = rand(1000)
```

```
julia> @btime f(x)
4.607 μs (9 allocations: 47.66 KiB)
```

```
julia> @btime f.(x)
938.481 ns (3 allocations: 7.95 KiB)
```

1.5.6. Paralelismo en Julia

La computación paralela es el uso simultáneo de múltiples recursos computacionales para resolver un problema computacional, es decir, es el uso de 2 o más procesadores para resolver una tarea.

La técnica se basa en el principio según el cual, algunas tareas se pueden dividir en partes más pequeñas que pueden ser resueltas simultáneamente.

Dentro de sus ventajas se encuentran:

- Resuelve problemas que no se pueden realizar con un solo CPU
- Resuelve problemas que no se pueden realizar en tiempo razonable
- Permite ejecutar problemas de un orden y complejidad mayor
- Ejecutar código mas rápido

Julia soporta estas cuatro categorías de programación concurrente y paralela:

Tareas asincrónicas o corrutinas

Julia Tasks permite suspender y reanudar cálculos para E / S, manejo de eventos, procesos de productor-consumidor y patrones similares. Las tareas se pueden sincronizar a través de operaciones como wait y fetch , y comunicarse a través de los Channel . Aunque no es estrictamente computación en paralelo por sí mismos, Julia le permite programar Task en varios subprocesos.

Multi-threading:

El subproceso múltiple de Julia brinda la capacidad de programar tareas simultáneamente en más de un subproceso o núcleo de CPU, compartiendo memoria. Esta suele ser la forma más fácil de obtener paralelismo en la propia PC o en un solo servidor grande de varios núcleos.

Cuando una función de subprocesos múltiples llama a otra función de subprocesos múltiples, Julia programará todos los subprocesos globalmente en los recursos disponibles, sin suscripciones en exceso. Se pueden usar los hilos para hacer creer al usuario (y a la propia PC) que si se puede hacer mas de una cosa al mismo tiempo.

Esto se logra de manera muy simple, en vez de realizar una tarea por completo, se divide la tarea en porciones (cada hilo se encarga de un aspecto concreto del programa), de modo que se va alternando entre porciones de tareas para que parezca que ambas se ejecutan al mismo tiempo.

El número de hilos corresponde de manera directa con el número de tareas que se pueden llevar a cabo de forma pseudoparalela (es decir, de forma 'simultánea').

Distributed computing (Computación distribuida)

La computación distribuida ejecuta múltiples procesos de Julia con espacios de memoria separados. Estos pueden estar en la misma computadora o en varias computadoras. La biblioteca estándar *Distributed* proporciona la capacidad para la ejecución remota de una función de Julia. Con este bloque de construcción básico, es posible construir muchos tipos diferentes de abstracciones de computación distribuida. Paquetes

como *DistributedArrays.jl* son un ejemplo de tal abstracción. Una implementación de la computación paralela de memoria distribuida es proporcionada por el módulo **Distributed** como parte de la biblioteca estándar enviada con Julia.

```
using Distributed
```

Julia proporciona un entorno de multiprocesamiento basado en el paso de mensajes para permitir que los programas se ejecuten en múltiples procesos en dominios de memoria separados a la vez.

La comunicación en Julia es generalmente "únilateral", lo que significa que el programador necesita manejar explícitamente sólo un proceso en una operación de dos procesos. Además, estas operaciones no suelen parecerse a ".envío de mensajes" recepción de mensajes", sino que se asemejan a operaciones de mayor nivel como las llamadas a funciones de usuario.

GESTIÓN DE LOS PROCESOS DE LOS TRABAJADORES

Las funciones `addprocs`, `rmprocs`, `workers` y otras están disponibles como medio programático para añadir, eliminar y consultar los procesos de un cluster.

El módulo `Distributed` debe ser cargado explícitamente en el proceso maestro antes de invocar `addprocs`. Está disponible automáticamente en los procesos de los trabajadores.

Al usar `addprocs`, se le dice a Julia que agregue n nuevos procesos de trabajo.

```
julia> addprocs(5)
5-element Array{Int32,1}:
 2
 3
 4
 5
 6
```

Muestra todos los procesos de los trabajadores

```
julia> workers()
5-element Array{Int32,1}:
 2
 3
 4
 5
 6
```

Elimina los workers 6 y 5

```
julia> rmprocs(6,5)
Task (done) @0xcdd9ad60
```

Quedaron esos workers

```
julia> workers()
3-element Array{Int32,1}:
 2
 3
 4
```

GPU Computing (Computación en la GPU):

El compilador de la GPU de Julia proporciona la capacidad de ejecutar código de Julia de forma nativa en las GPU. Existe un rico ecosistema de paquetes de Julia que se dirigen a las GPU. El sitio web JuliaGPU.org proporciona una lista de capacidades, GPU compatibles, paquetes relacionados y documentación.

Capítulo 2

Unidades de procesamiento gráfico

2.1. Necesidad de las GPU

La Unidad de procesamiento gráfico (GPU por sus siglas en inglés: Graphics Processing Unit) es sin duda, uno de los componentes más importantes de todos los que se utilizan actualmente para ya sea el armado de computadoras o algún rendimiento que se busque obtener de cierto proceso.

Es muy importante tomar en cuenta que el concepto de GPU que se maneja hoy en día no es exactamente el mismo que se tenía en la década de los noventa, y a su vez, el concepto de aquella época, tampoco encaja con el que se utilizaba, por ejemplo, en los años ochenta. Esto se debe a que la arquitectura, la unidad de procesamiento gráfico y su forma de trabajar, esto ha ido cambiando para adaptarse a la evolución del sector. En los años setenta y ochenta, los gráficos en 3D tenían un coste tan alto en términos de rendimiento que eran algo prohibitivo, y casi todos los adaptadores gráficos de aquella época estaban limitados al 2D. se podían utilizar técnicas como el blitting y el dibujado de sprites, estas se combinaban con diferentes efectos para dar forma a trabajos que, en algunos casos, alcanzaban una complejidad importante.

Por tanto se puede definir a la GPU como una unidad especializada que se ocupa de sacar adelante todo lo relacionado con la carga de trabajo gráfico que debe procesar un sistema. A diferencia del procesador, que realiza tareas de propósito general, la GPU se especializa en tareas que son necesarias para crear elementos gráficos, tanto en 2D como en 3D, y está capacitada para realizar una alta cantidad de operaciones de coma flotante por segundo. Un procesador o CPU carece de ese nivel de especialización, y no está capacitado para trabajar con elementos gráficos. La carga de trabajo que debe afrontar una GPU está formada por tareas muy importantes, que incluyen desde la representación visual más básica, como el escritorio de trabajo de la PC, hasta la ejecución de gráficos 3D avanzados en juegos de última generación, pasando por la decodificación y la aceleración de videos en diferentes formatos, el tratamiento del color y las funciones de posprocesado asociados a imágenes, videos y otros contenidos multimedia.

Como podemos ver, el papel de la GPU es muy importante, y aunque su forma de trabajar presenta similitudes cuando se compara con la CPU, al final se debe tener muy en claro que es un componente muy distinto. Una GPU recibe datos e instrucciones de la CPU, cuenta con su propia memoria para guardar determinados elementos geometría, texturas, shaders, etc, a los que puede acceder cuando lo necesita, sin tener que volver a procesarlos, pero su trabajo se limita a las tareas gráficas, y para afrontarlas recurre un enorme grado de paralelizado.

Una CPU de alto rendimiento actual puede contar con ocho núcleos. Esos núcleos le permiten paralelizar ocho procesos distintos. Pues bien, una GPU, como la RTX 3080, cuenta con 8.704 shaders o núcleos CUDA, es decir, tiene miles de pequeños núcleos, lo que permite paralelizar cargas de trabajo grandes y complejas, y sacarlas adelante de una forma más eficiente.

En resumen, una GPU es una unidad de procesamiento gráfico con una alta capacidad de paralelizado, capaz de trabajar y de procesar gráficos, y de convertir información y datos en elementos visibles por el usuario, pero que también de sacar adelante tareas que requieran de la realización de una gran cantidad de operaciones concurrentes en paralelo.

La GPU es el motor gráfico de cualquier equipo. Sin ella, no se podría hacer algo tan simple representar el escritorio de Windows 10, por ejemplo, y tampoco se podría ejecutar juegos 3D o disfrutar de contenidos multimedia en alta resolución. Es importante tener en cuenta que una GPU solo desarrolla todo su potencial en aplicaciones y tareas que estén preparadas para aprovechar la aceleración a través de GPU, es decir, para paralelizarse en niveles muy elevados.

Sin embargo, su rendimiento y su eficiencia varían en función de la arquitectura y del proceso de fabricación que utilice.

Esto quiere decir que una GPU que tenga un mayor número de shaders no tiene por qué ser mejor que otra que cuente con un menor número de aquellos. Por ejemplo, la GTX 780 Ti tiene la friolera de 2.880 shaders, pero utiliza una arquitectura obsoleta Kepler, y está fabricada en proceso de 28 nm, por lo que rinde menos que una GTX 1060 basada en Pascal, que suma 1.280 shaders, y es menos eficiente que esta, que viene fabricada en proceso de 16 nm.

Con el lanzamiento de cada nueva arquitectura, tanto NVIDIA como AMD han introducido mejoras importantes a nivel de rendimiento que no siempre han estado ligadas a un aumento del número máximo de shaders.

2.2. ¿Qué son las GPU's

Una unidad de procesamiento gráfico (GPU por sus siglas en inglés: Graphics Processing Unit) representa el corazón de la tarjeta gráfica, algo similar a lo que hace la CPU con la PC. Además, también funciona como el cerebro de la tarjeta gráfica, ya que se encarga de realizar todos los cálculos complejos lo que permite tener gráficos de calidad y debidamente procesados en la computadora. Además, aligerar la carga de trabajo en el procesador (CPU) en aplicaciones interactivas 3D o videojuegos. Así, mientras gran parte de lo relacionado con los gráficos se procesa en la GPU, la unidad central de procesamiento (CPU) puede dedicarse a otro tipo de cálculos (como la inteligencia artificial o los cálculos mecánicos en el caso de los videojuegos).

La carga de trabajo que debe afrontar una GPU está formada por tareas muy importantes, que incluyen desde la representación visual más básica, como el escritorio de la PC, hasta la ejecución de gráficos 3D avanzados en juegos de última generación, pasando por la decodificación y la aceleración de videos en diferentes formatos, el tratamiento del color y las funciones de posprocesado asociados a imágenes, videos y otros contenidos multimedia.

GPU y tarjeta gráfica son dos términos que a veces se usan indistintamente. Sin embargo, existen algunas distinciones importantes entre los dos. La principal diferencia es que la GPU es una unidad específica dentro de una tarjeta gráfica.

La GPU es la que realiza el procesamiento real de imágenes y gráficos. Una tarjeta gráfica es lo que presenta imágenes a la unidad de visualización.

Algunos ejemplos de casos de uso de GPU:

- Las GPU pueden acelerar la representación de aplicaciones de gráficos 2D y 3D en tiempo real.
- La edición de video y la creación de contenido de video han mejorado con las GPU. Los editores de video y los diseñadores gráficos, por ejemplo, pueden usar el procesamiento paralelo de una GPU para acelerar la reproducción de video y gráficos de alta definición.
- Los gráficos de los videojuegos se han vuelto más intensivos desde el punto de vista informático, por lo que para mantenerse al día con las tecnologías de visualización, como 4K y altas frecuencias de actualización, se ha puesto énfasis en las GPU de alto rendimiento.
- Las GPU pueden compartir el trabajo de las CPU y entrenar redes neuronales de aprendizaje profundo para aplicaciones de IA. Cada nodo de una red neuronal realiza cálculos como parte de un modelo analítico.

La GPU consiste en un conjunto de multiprocesadores SM (streaming multiprocessor), cada uno de los cuales es capaz de soportar miles de hilos concurrentes co-residentes. Cada SM de la GPU es un conjunto de procesadores. En cada ciclo de reloj, un multiprocesador ejecuta la misma instrucción en un grupo de hilos denominado warp.

Su diseño arquitectónico se basa en:

- Ejecución SIMT (Single Instruction Multiple Threads)

El hardware maneja la divergencia automáticamente

- Multihilo por hardware

Asignación de recursos HW y programación de hilos

El hardware se basa en los hilos para ocultar la latencia

El cambio de contexto es (básicamente) gratuito

Al inicio, la programación de la GPU se realizaba con llamadas a servicios de interrupción de la BIOS. Tras esto, la programación de la GPU se empezó a hacer en el lenguaje ensamblador específico a cada modelo. Posteriormente, se introdujo un nivel más entre el hardware y el software, con la creación de interfaces de programación de aplicaciones (API) específicas para gráficos, que proporcionaron un lenguaje más homogéneo para los modelos existentes en el mercado. La primera API usada ampliamente fue el estándar abierto OpenGL (Open Graphics Language), tras el cual Microsoft desarrolló DirectX.

Tras el desarrollo de estas API, se decidió crear un lenguaje más próximo al natural utilizado por el programador.

2.3. Diferencia entre la GPU y la CPU

Un procesador (CPU) realiza operaciones de propósito general. Para ello, utiliza sus diferentes núcleos, almacena instrucciones y datos que necesita en su memoria caché y guarda operaciones resueltas en la memoria RAM, lo que le permite volver a acceder a esas operaciones cuando las necesita sin tener que volver a procesarlas.

Una GPU lo hace de manera similar. Recibe datos e instrucciones de la CPU, cuenta con su propia memoria para guardar determinados elementos (geometría, texturas, shaders, etc), a los que puede acceder cuando lo necesita, sin tener que volver a procesarlos. Su trabajo se limita a las tareas gráficas y utiliza un enorme grado de paralelismo.

Es decir, mientras una CPU tiene unos cuantos núcleos optimizados para el procesamiento en serie secuencial,

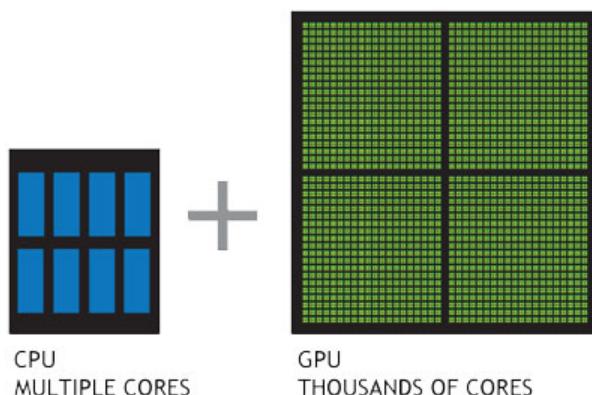


Figura 2.1: Núcleos de la CPU vs núcleos de la GPU

una GPU tiene una arquitectura en paralelo enorme que consiste de miles de núcleos más pequeños y eficaces que se diseñaron para resolver varias tareas al mismo tiempo.

Las GPU tienen miles de núcleos para procesar cargas de trabajo en paralelo de forma eficiente.

2.4. Elementos que integran a la GPU

Todas las GPU dentro de una misma arquitectura independientemente de sus diferentes tamaños comparten todos los elementos comunes que definen a la misma, pero a diferentes niveles de potencia. Se detallan a continuación:

- **Procesador de comandos:** Se encarga de leer cada una de las instrucciones que son enviadas por parte de la CPU. De esta forma podrá generar los gráficos que sean necesarios o los cálculos complejos necesarios.
- **Unidad de texturizado:** Se encarga de aplicar una imagen sobre una superficie, esto para simular una textura o un color realistas. Básicamente le da textura a los píxeles procesados.
- **Unidad de rasterizado:** Esta unidad realiza la transformación del espacio tridimensional basado en vértices a uno bidimensional basado en píxeles.

- **Unidad shader:** Son capaces de ejecutar cada uno de los programas que manipulan primitivas de los gráficos en tiempo real.
- **Unidad de intersección:** Se encarga de calcular la intersección de los rayos que están en las escenas de los objetos. Es fundamental para el trazado de rayos (Ray Tracing).
- **Unidad de teselación:** Unidad que subdivide los vértices de los objetos para darles un aspecto más redondeada y pulido.
- **Raster output (ROPs):** Se encarga de dibujar los píxeles finales sobre el búfer de la imagen. La caché y este son los únicos elementos que puede escribir dentro de la memoria VRAM. Actúa en dos etapas durante el pipeline, la primera en la etapa de rasterización donde genera el Z-Búfer, un búfer de imagen que indica a qué distancia está cada elemento de la escena respecto a la cámara. La segunda tras el texturizado donde genera los búferes de color en formato RGBA.
- **CODEC de vídeo:** Este elemento se encarga de procesar y decodificar los videos en diferentes formatos. De esta forma podrá reproducirlos, generar videos e incluso cambiarlos de formatos sin problema.
- **Interfaz de memoria:** Esto es lo que permite que el procesador gráfico pueda leer su propia memoria que es conocida como VRAM.
- **DMA:** Este es el componente que le permite la lectura de la memoria RAM principal del sistema.
- **Memoria gráfica (VRAM):** Aunque no se integra en la GPU, juega un papel muy importante. Su función es muy similar a la que ejerce la memoria RAM con respecto a la CPU. La GPU utiliza la memoria gráfica para guardar determinados elementos y datos gráficos que ya ha procesado, para poder recurrir a ellos cuando los necesite sin tener que volver a procesarlos, es decir, sin tener que repetir ciclos de trabajo. Texturas, geometría, shaders y otros elementos se guardan en la memoria gráfica. Una cantidad insuficiente de memoria gráfica puede limitar el rendimiento de una GPU y su frecuencia de trabajo también influye, ya que la memoria gráfica más lenta ofrece unas velocidades de acceso (escritura y lectura) inferiores. Es una memoria de tipo local por lo que generalmente se encuentra fuera del alcance de la CPU. Es decir, no puede ser usada por el procesador central de la PC para ejecutar programas.
- **Núcleos RT:** Se encargan de acelerar el procesado de la carga de trabajo relacionada con el trazado de rayos. Estos núcleos fueron introducidos en 2018, y se centran en calcular las intersecciones transversales BVH, las intersecciones rayo-triángulo y a las intersecciones de delimitadoras de cuadro.
- **Núcleos tensor:** Se utilizan para acelerar la IA, son relativamente nuevos. Se centran en la inferencia, IA y aprendizaje profundo, para mejorar la imagen mediante tecnologías como FSR de AMD, DLSS de NVIDIA, etc.

2.5. Arquitectura de la GPU

Las GPUs están organizadas de tal manera que los diferentes conjuntos que las componen se encuentran en muchos casos unos dentro de otros.

Conjunto A en la organización de una GPU: las unidades shader

El primero de los conjuntos son las unidades shader. Por ellas mismas son procesadores, pero al contrario que las CPU no están pensadas para el paralelismo a partir de las instrucciones (Instruction Level Parallelism, ILP), sino a partir de los hilos de ejecución (Parallelism task level, TLP). Independientemente de si es una GPU de AMD, NVIDIA, Intel o cualquier otra marca, toda GPU se compone de:

- Unidades SIMD (Single Instruction/Multiple Data) y sus registros. Las unidades SIMD son un tipo de unidad de ejecución que está pensada para ejecutar la misma instrucción a varios datos al mismo tiempo.
- Unidades Escalares y sus registros.
- Planificador

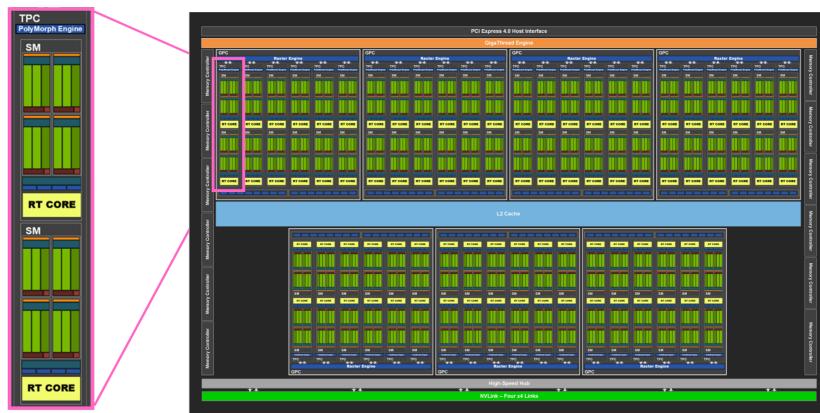


Figura 2.2: Conjunto A: Unidades shader

- Memoria Local Compartida
- Unidad de Filtrado de Texturas
- Caché de datos y/o texturas de primer nivel
- Unidades Load/Store para mover datos desde y hacia la caché y la memoria compartida.
- Unidad de Intersección de Rayos.
- Arrays Sistólicos o unidades tensor
- Export Bus que exporta datos hacia fuera del Conjunto A y hacia los diferentes componentes del Conjunto B.

Conjunto B en la organización de una GPU: Shader Array/Shader Engine/GPC



Figura 2.3: Conjunto B: Shader Array/Shader Engine/GPC

El Conjunto B incluye en su interior al conjunto A en su interior, pero de entrada añade las cachés de instrucciones y de constantes. En las GPU al igual que las CPU la caché de primer nivel se encuentran divididas en dos partes, una para datos y la otra para instrucciones. La diferencia es que en el caso de las GPU la caché de instrucciones se encuentra fuera de las unidades shader y por tanto se encuentran en el

conjunto B.

El conjunto B en la organización de una GPU por lo tanto incluye una serie de unidades shader, que se comunican entre sí a través de la interfaz de comunicación en común entre ellas, lo que les permite comunicarse entre sí. Por otro lado las diferentes unidades shader no están solas en el Conjunto B, ya que es aquí donde hay varias unidades de función fija (de rasterización, ROPS) para el renderizado de los gráficos.

Conjunto C en la arquitectura de una GPU



Figura 2.4: Conjunto C: Arquitectura general de la GPU

La GPU sin los aceleradores, se compone de los siguientes componentes:

- Varios Conjuntos B en su interior.
- Memoria Global Compartida: Un Scratchpad y por tanto fuera de la jerarquía de caches para comunicar los Conjuntos B entre si.
- Unidad Geométrica: Tiene la capacidad de leer los punteros a la RAM que apuntan a la geometría de la escena, con ello es posible eliminar geometría no visible o superflua para que no se renderice inútilmente en el fotograma.
- Procesadores de Comandos (Gráficos y Computación)
- Caché de Último Nivel: Todos los elementos de la GPU son clientes de esta caché por lo que tiene que tener un anillo de comunicación inmenso, todos los componentes del Conjunto B tienen contacto directo con la caché L2 así como todos los componentes del propio Conjunto C.

La caché de Último Nivel (Last Level Cache, LLC) es importante ya que es la cache que da coherencia entre todos los elementos del Conjunto C entre si incluyendo obviamente los Conjuntos B dentro del mismo. Ademas permite no sobre-saturar el controlador de memoria externa ya que con esto es la propia LLC junto a la o las unidades MMU de la GPU las que se encargan de hacer la captación de instrucciones y datos desde la RAM.

2.5.1. Arquitecturas NVIDIA

En esta sección se dara una breve descripción sobre algunas arquitecturas de NVIDIA.

Arquitectura NVIDIA Tesla

Tesla es la primera micro arquitectura de Nvidia que implementa el modelo de shaders unificados. Los drivers soportan arquitectura Direct3D 10 Shader Model 4.0 / OpenGL 2.1(drivers posteriores tienen soporte OpenGL 3.3). El diseño es el mayor cambio en la funcionalidad y capacidad de las GPU de Nvidia, El cambio más evidente es la separación de sus unidades funcionales (pixel shaders, vertex shaders) dentro de

las anteriores GPU a una colección homogénea de los procesadores puntos flotantes (llamados "procesadores de flujo") que pueden llevar cabo un conjunto más de tareas universales.

Se usaron con las tarjetas GeForce 8 Series, GeForce 9 Series, GeForce 100 Series, GeForce 200 Series, y GeForce 300 Series en GPUs manufacturadas en 90 nm, 80 nm, 65 nm, y 55 nm. También se encuentran en el uso de GeForce 405, y en el mercado de trabajo de Quadro FX, Quadro x000, Quadro NVS series, y Nvidia Tesla módulos informáticos. Tesla reemplazo la vieja Segmentación fija de la micro arquitectura y compite directamente con la primera micro arquitectura de shader unificado de AMD llamada escala tera. Tesla fue el predecesor de Fermi.

Arquitectura Fermi

Fermi ofrece una serie de características que aceleran el desempeño de una gama de aplicaciones más amplia. Oak Ridge National Laboratory, quien se unió a la conferencia de prensa de Nvidia, anunció planes para una nueva super computadora que utilizará Nvidia GPUs basadas en la arquitectura Fermi.

Como la base para la familia de GPUs de próxima generación de Nvidia denominadas GeForce, Quadro y Tesla. Fermi incluye un grupo de nuevas tecnologías que son características imprescindibles para el área de cómputo, incluyendo:

- C++, complementando el actual soporte para C, Fortran, Java, Python, OpenCL y DirectCompute.
- – ECC, requerimiento crítico para centros de datos y centros de super cómputo que implementan GPUs a gran escala.
- – 512 CUDA Cores que incluyen el nuevo estándar de punto flotante IEEE 754-2008, que superan incluso a las CPUs más avanzadas.
- Nvidia Parallel DataCache, la primera jerarquía de caché real en una GPU que acelera algoritmos como solucionadores físicos, seguimiento de rayos, y multiplicación de matriz escasa donde las direcciones de los datos no son conocidos de antemano.
- Nvidia GigaThread Engine con soporte para la actual ejecución de kernels, donde los diferentes kernels del mismo contexto de las aplicaciones pueden ejecutarse en la GPU al mismo tiempo.

GPU con arquitectura Fermi:

- GF100: se utiliza en la GTX 480
- GF104: se utiliza en la GTX 460
- GF106: se usa en algunos GT 440, en el GTS 450
- GF108: se usa en GT 430, algunos GT 440,
- GT 530, algunos GT 630, algunos GT 730
- GF110: se utiliza en GTX 560 Ti 448 Core, GTX 570, GTX 580 y GTX 590
- GF114: se utiliza en GTX 560 Ti y GTX 560
- GF116: se utiliza en la GTX 550 Ti, GeForce GT 640
- GF119: se utiliza en GT 520, GeForce 605, GeForce GT 610, GeForce GT 620
- GF117: grabado de 28 nm

Arquitectura NVIDIA Kepler

Se trata de la arquitectura Fermi, por lo que es un inteligente dividido en GPC o Clúster de procesamiento de gráficos que son el equivalente a un corazón de microprocesador a menos que estén desprovistos de caché, constan de uno, dos o tres SMX y contienen 8 unidades de ROP. Para admitir los núcleos CUDA, que son las unidades de cálculo, el chip ofrece dos niveles de memoria caché (L1 y L2). El chip está equipado con 512 KB de memoria caché L2, con una mayor velocidad en comparación con Fermi.

Un SMX , es la gran novedad introducida por Kepler, es el reemplazo del SM de Fermi. Tiene capacidad

para 192 núcleos CUDA , o cálculos de unidades, la versión más avanzada frente a 32 núcleos CUDA con el SM de Fermi . Por otro lado, el SMX abandona el sistema de cronometraje doble de Fermi que duplicó la frecuencia SM y por lo tanto las unidades de cálculo. Cada SMX tiene 64 KB de memoria compartida y 16 unidades de textura.

La arquitectura Kepler se utiliza en múltiples GPU:

- el GK 104, que se utiliza en GTX 680, GTX 690, GTX 670, GTX 660, GTX 660ti, GTX 760, GTX 770, Tesla K10, Tesla K8;
- la GK 106, que se utiliza en la GTX 660, está compuesta por 5 SMX;
- la GK 107, que se utiliza en tarjetas móviles de NVidia , así como en tarjetas de nivel de entrada, comenzando con la GT 640;
- la GK 110, que tiene 15 SMXs más potentes y 1,5 MB de memoria caché , se utiliza en el Tesla K20, GeForce Titán, GeForce GTX 780 y GeForce GTX 780 Ti

En 2013, todos los chips Kepler se grabaron a 28 nm .

Arquitectura NVIDIA Maxwell

Los primeros productos que contaban con Maxwell en salir al mercado , fueron la Geforce GTX 750 y GTX 750 Ti. Ambos fueron lanzados el 18 de febrero de 2014 y utilizaban el chip GM107. Las primeras series de Geforce 700 aún continuaban utilizando Kepler, con chips GK1XX. Las GPU GM10X también fueron utilizadas en la serie Geforce 800M y Quadro Kxxx.

Una segunda generación de productos Maxwell fue introducida el 18 de septiembre de 2014, con la Geforce GTX 970 y la Geforce GTX 980, seguido por la Geforce GTX 960 el 22 de enero de 2015, durante ese mismo año se lanzaron la Geforce GTX Titan X en marzo 17 y la Geforce GTX 980 Ti el primero de junio. Estas GPU contaban con la serie de chip GM20x.

Maxwell introdujo un diseño completamente nuevo para el Streaming Multiprocesador (SM) que dramáticamente mejora la eficiencia de energía, además la sexta y séptima generación de PureVideo (NVIDIA) y la versión 5.2 de CUDA.

Arquitectura NVIDIA Pascal

La arquitectura de NVIDIA Pascal se basa en cinco avances tecnológicos, lo que permite una nueva plataforma de computación que interrumpe el pensamiento convencional desde el escritorio hasta el data center.

NVIDIA comienza a utilizar la tecnología HBM por primera vez en esta arquitectura dotando los modelos de más alta gama con HBM 2, mientras que los demás permanecen con memorias GDDR5X y GDDR5. También se estrena el NV Link, una tecnología elaborada entre NVIDIA e IBM para dotar a tarjetas gráficas profesionales de un bus distinto al PCI-e que otorga una velocidad 12 veces mayor entre GPU y CPU.

Arquitectura NVIDIA Volta

Volta se creó con nueva tecnología de gráficos y un hardware mas rápido. Cuenta con una arquitectura de próxima generación. Equipada con 640 Tensor Cores, Volta ofrece más de 100 Teraflops por segundo (TFLOPS) de rendimiento de deep learning, más de un aumento de 5 veces en comparación con la arquitectura NVIDIA Pascal. Cuenta con más de 21 billones de transistores y combina los NVIDIA® CUDA® y Tensor Cores para ofrecer el rendimiento de un supercomputador IA en una GPU.

2.5.2. Arquitecturas AMD

AMD es una compañía que desarrolla procesadores de computación y productos tecnológicos similares de consumo. Sus productos principales incluyen microprocesadores, chipsets para placas base, circuitos integrados auxiliares, procesadores embebidos y procesadores gráficos para servidores, estaciones de trabajo, computadores personales y aplicaciones para sistemas embebidos.

Dentro de sus muchas aplicaciones AMD ha desarrollado tarjetas gráficas a partir del 2006 a partir de la

compra de ATI el segundo mayor fabricante de tarjetas gráficas del mundo, y rival directo de Nvidia durante muchos años. Graphics Core Next, la primera arquitectura gráfica completamente de AMD, Graphics Core Next es el nombre en clave para una serie de microarquitecturas y un conjunto de instrucciones. Esta arquitectura es la sucesora de la anterior TeraScale creada por ATI. El primer producto basado en GCN, la Radeon HD 7970 se lanzó en 2011.

GCN es una microarquitectura RISC SIMD que contrasta con la arquitectura VLIW SIMD de TeraScale. GCN requiere muchos más transistores que TeraScale, pero ofrece ventajas para el cálculo de GPGPU, hace el compilador más simple y también conducir a una mejor utilización de los recursos. GCN está fabricado en los procesos a 28 y 14 nm, disponibles en los modelos seleccionados de las series Radeon HD 7000, HD 8000, R 200, R 300, RX 400 y RX 500 de tarjetas gráficas AMD Radeon. La arquitectura GCN también se utiliza en el núcleo gráfico de APU de PlayStation 4 y Xbox One.

En los últimos años AMD ha desarrollado otra arquitectura de GPU la RDNA la cuál se ha desarrollado de

TARJETAS GRÁFICAS AMD POLARIS Y VEGA							
Tarjeta gráfica	Compute Units/Shaders	Frecuencia de reloj base/turbo	Cantidad de memoria	Interfaz de memoria	Tipo de memoria	Ancho de banda de memoria	TDP
AMD Radeon RX Vega 56	56/3.584	1156/1471 MHz	8 GB	2.048 bits	HBM2	410 GB/s	210W
AMD Radeon RX Vega 64	64/4.096	1247/1546 MHz	8 GB	2.048 bits	HBM2	483,8 GB/s	295W
AMD Radeon RX 550	8/512	1183 MHz	4 GB	128 bits	GDDR5	112 GB/s	50W
AMD Radeon RX 560	16/1.024	1175/1275 MHz	4 GB	128 bits	GDDR5	112 GB/s	80W
AMD Radeon RX 570	32/2.048	1168/1244 MHz	4 GB	256 bits	GDDR5	224 GB/s	150W
AMD Radeon RX 580	36/2.304	1257/1340 MHz	8 GB	256 bits	GDDR5	256 GB/s	180W

Figura 2.5: Tarjetas Gráficas AMD

mejor manera que las GCN, RDNA 1 se estrenó en la familia de tarjetas gráficas AMD RX 5000, con nombre en clave Navi. Esta marcó un punto de inflexión bastante claro en el rumbo que seguía su división gráfica. Por lo que Navi supuso grandes mejoras en eficiencia energética y rendimiento. Se puede apreciar en la gráfica

TARJETAS GRÁFICAS AMD NAVI (MODELOS MÁS RELEVANTES)							
Tarjeta gráfica	Unidades de cómputo	Frecuencia de reloj base/turbo	Cantidad de memoria	Interfaz de memoria	Tipo de memoria	Ancho de banda de memoria	TBP
AMD Radeon RX 5700 XT	40	1605/1905MHz	8GB	256 bits	GDDR6	448GB/s	225W
AMD Radeon RX 5700	36	1465/1725MHz	8GB	256 bits	GDDR6	448GB/s	180W
AMD Radeon RX 5600 XT	36	1375/1750MHz	6GB	192 bits	GDDR6	288-336GB/s	150-160W
AMD Radeon RX 5500 XT	22	1717/1845MHz	4/8	128 bits	GDDR6	224GB/s	130W

Figura 2.6: Tarjetas Gráficas AMD Navi

anterior que el modelo más alto de Navi fue la RX 5700 XT. Esta es una gráfica que compite mayormente con la RTX 2070 o la RTX 2060 SUPER, o en algún caso la 2070 SUPER de la marca NVIDIA. En cambio, se queda más o menos un 25 por ciento por detrás de lo mejor de NVIDIA. Esto pone de evidencia que en esta generación todavía no fueron capaces de superarles.

Esto dio paso a que AMD optara por la RDNA 2 la cuál es una versión levemente mejorada de RDNA y no un cambio menos radical, por lo que AMD optara por ir lanzando mejoras continuas sobre una misma arquitectura. Se dice que AMD lanzó RDNA como una solución temporal mientras terminaban de pulir RDNA 2 que es la versión ya terminada de la arquitectura y totalmente compatible con DirectX 12 Ultimate. Si hablamos en términos de computación, RDNA 2 no tiene ninguna ventaja respecto a RDNA y las mejoras se han dado más bien en elementos ajenos a la parte encargada de ejecutar los shaders.

Capítulo 3

CUDA-C



3.1. Preámbulo

A lo largo este capítulo se estará viendo a gran detalle todo lo referente a CUDA, se abarcarán conceptos fundamentales para entender este paradigma de programación, así como distintos ejemplos que sean de ayuda en cada tema, con lo que se pretende lograr que el lector tenga bien sentadas las bases de lo que será CUDA, en este capítulo se repasarán conceptos fundamentales como los núcleos de CUDA, Kernel, Hilos, los tipos de memoria que nos ofrece CUDA, están presentes.

3.2. Introducción a CUDA-C.

Arquitectura Unificada de Dispositivos de Cómputo (Compute Unified Device Architecture) es una extensión de la programación en C o C++, CUDA es un lenguaje de programación que utiliza la unidad de procesamiento gráfico (GPU), utilizando la computación paralela y un modelo API, desarrollado por NVIDIA.

CUDA permite que los cálculos se realicen en paralelo al mismo tiempo de que la velocidad no se ve afectada, con lo que se busca aprovechar la potencia de la GPU NVIDIA para realizar tareas informáticas comunes, tales como el procesamiento de matrices y otras operaciones de álgebra lineal, en lugar de solo realizar cálculos gráficos.

CUDA tiene muchas ventajas ya que las GPU cuentan con Unidades Lógicas Aritméticas diminutas a comparación con las que tienen las CPU, esto con la finalidad de realizar múltiples cálculos paralelos, como por

ejemplo el poder calcular el color de cada pixel en la pantalla, etc.

La compatibilidad de CUDA es muy extensa ya que esta es compatible con Windows, Linux y MAC OS, al igual que es una plataforma en la que podemos encontrar en tarjetas graficas de las gamas: Geforce, Quadro y Tesla, ya que estan diseñadas para ser de escritorio, portatiles, profesionales al igual que para superordenadores.

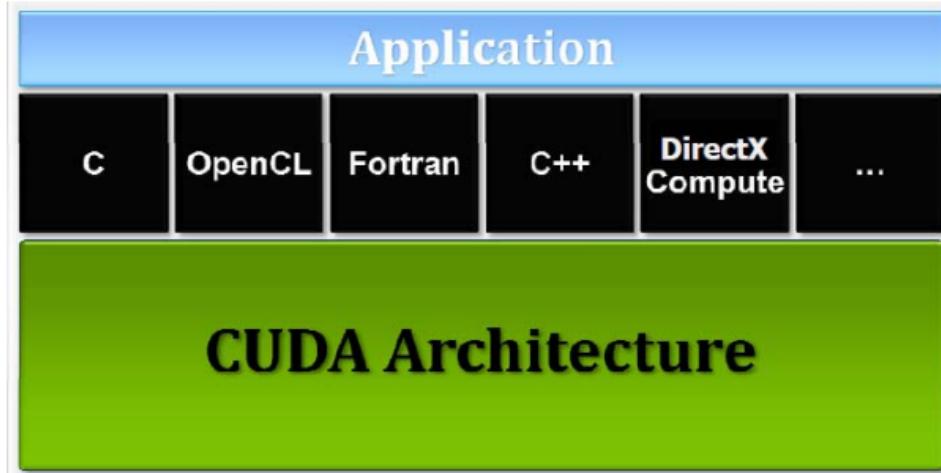


Figura 3.1: Soporte de CUDA

3.2.1. ¿Cómo funciona CUDA?

En CUDA las GPU ejecutan un grupo de tareas llamada “kernel” a la vez lo que permite un procesamiento más rápido de información así como la ejecución paralela de distintas tareas. En CUDA se cuentan con varios núcleos donde cada núcleo consta de bloques los cuales son independientes de la ALU, a su vez cada bloque tiene subprocessos los cuales representan los niveles de cálculo y cada subprocesso de cada bloque trabajan en conjunto para calcular un valor, es posible que los subprocessos en el mismo bloque puedan compartir memoria. En CUDA la información puede ser enviada desde la CPU a la GPU y para cada subprocesso la memoria local es la más rápida, seguida de la memoria compartida, la global, estática y la memoria de textura es la más lenta de todas. Durante la distribución de trabajo que realiza la GPU cada subprocesso conoce las coordenadas x e y del bloque en donde está al igual que las coordenadas en las que se encuentra en el bloque, esto con el fin de calcular una ID de un subprocesso única para cada subprocesso.

3.2.2. Modelo de programación CUDA

Se sabe que la programación paralela requiere un paradigma muy amplio y distinto a los modelos de programación convencionales, el código se ejecuta en miles de núcleos al mismo tiempo, esto con la finalidad de aprovechar al máximo el paradigma que ofrece CUDA. Dentro de este paradigma se encuentran conceptos importantes como lo son los hilos, núcleos, subprocessos, kernel, etc. Lo cual hace la comprensión de mejor manera de cómo es que la GPU ejecutará las tareas que se pongan. Se debe tomar en cuenta que, para problemas grandes puede solicitar subprocessos muy por encima de la cantidad de núcleos, se debe comprender que la cantidad de subprocessos no es la cantidad de núcleos en una GPU, la GPU comenzará a ejecutar tantos subprocessos como pueda físicamente y ejecutará más subprocessos a medida que se completen los subprocessos que ya se están ejecutando.

Un hilo es una unidad lógica mientras que el núcleo es una unidad física, desde el punto de vista del programador todos los hilos que se solicitan, se ejecutan al mismo tiempo, por lo tanto CUDA se encargará de la programación y el orden de los subprocessos en lugar del programador.

Una de las ventajas que ofrece CUDA es quitarle al programador los detalles de la paralelización, CUDA lo único que necesita es dividir la tarea para que cada subprocesso tenga una subtarea definida y después de eso solicite tantos subprocessos como sea necesario para resolver el problema solicitado.

3.2.3. Núcleos de CUDA

Los NVIDIA CUDA Cores o núcleos CUDA son procesadores paralelos, los cuales se encargan de procesar todos los datos que entran y salen de la GPU, es capaz de realizar cálculos gráficos los cuales pueden ser consultados por el usuario final, se encuentran dentro de la GPU y de sus principales tareas consiste en el renderizado de objetos 3D, dibujar modelos, comprende y resuelve la iluminación de escenas creadas, etc.

La computación paralela permite a los núcleos trabajar al mismo tiempo para así completar una misma tarea, esto es de mucha ayuda para la CPU cuando maneja datos ya que ambos núcleos CPU y GPU trabajan en conjunto para completar una o varias tareas al mismo tiempo sin perder ineficiencia.

3.3. Hola Mundo con CUDA

Uno de los programas más comunes al empezar a programar en cualquier lenguaje es un "Hola Mundo" que te abre el paradigma de la programación de una manera tan sencilla como imprimir un simple mensaje en pantalla, en CUDA esto no es la excepción, si bien la sintaxis está escrita en C++, la ejecución de este programa será la misma:

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <cuda_runtime.h>
4 // rutina principal ejecutada en el host
5 int main(int argc, char** argv)
6 {
7     // cuerpo del programa
8     printf("Hola, mundo!!\n");
9     // salida del programa
10    printf("\npulsa INTRO para finalizar... ");
11    fflush(stdin);
12    char tecla = getchar();
13    return 0;
14 }
```

Listing 3.1: Hola Mundo en CUDA

3.3.1. Kernel

Un kernel es el código que se ejecuta en el dispositivo, la función que ejecutan los diferentes flujos durante la fase paralela. Este kernel se ejecuta de forma paralela dentro de la GPU como un conjunto de hilos (threads) y que el programador organiza dentro de una jerarquía en la que pueden agruparse en bloques (blocks), y que a su vez se pueden distribuir formando una malla (grid). Por conveniencia los bloques y las mallas pueden tener una, dos o tres dimensiones.

Existen multitud de situaciones en las que los datos con los que se trabaja poseen de forma natural una estructura de malla, pero en general, descomponer los datos en una jerarquía de hilos no es una tarea fácil. Así pues, un bloque de hilos es un conjunto de hilos concurrentes que pueden cooperar entre ellos a través de mecanismos de sincronización y compartir accesos a un espacio de memoria exclusivo de cada bloque. Y una malla es un conjunto de bloques que pueden ser ejecutados independientemente y que por lo tanto pueden ser lanzados en paralelo en los Streaming Multiprocessors (SM). Dentro del código de CUDA un Kernel se define de la siguiente manera:

```

1 Definicion del kernel
2 __global__ void f(int a, int b, int c)
3 {
4 }
```

Listing 3.2: Definición del Kernel

En un ejemplo si tenemos una función denominada *f* y se quiere calcular la diferencia entre A y B y el resultado de esta operación se almacene en una variable C, se podría definir de la siguiente manera:

```

1 __global__ void f(int* A, int* B, int* C)
2 {
3     int i = threadIdx.x;
4     C[i] = A[i] - B[i];
```

5 }

Listing 3.3: Calculo de los vectores

Como sabemos los hilos son parte importante de la programación en paralelo, en esta función se ejecutaría una vez en cada hilo, reduciendo el tiempo total de ejecución en gran medida, y dividiendo su complejidad, por una constante directamente relacionada con el número de procesadores disponibles.

Cuando se lanza un kernel se crea una malla de hilos donde todos ellos ejecutan el mismo programa. El kernel especifica las instrucciones que van a ser ejecutadas por cada hilo individual y se pueden lanzar todos los hilos en un único bloque, lanzar varios bloques con un solo hilo cada uno, o lanzar varios bloques con varios hilos en cada bloque. De esta forma, el GPU se encarga de ejecutar simultáneamente tantas copias del kernel como hilos se tenga en ejecución. A cada hilo se le asignan sus propios datos de forma que cada copia realice la misma operación pero con datos diferentes, aprovechando así el paralelismo que ofrecen las GPUs.

3.3.2. ¿Cómo es la ejecución del Kernel?

La memoria del procesador principal y del dispositivo son espacios de memoria completamente separados, lo que permite la computación simultánea tanto en la CPU como en la GPU sin competir por los recursos de memoria. Para ejecutar un kernel en el dispositivo GPU se siguen los siguientes pasos:

1. Reservar memoria en el dispositivo
2. Transferir los datos necesarios del procesador principal al espacio de memoria asignado al dispositivo.
3. Invocar la ejecución del kernel en cuestión.
4. Transferir los resultados del dispositivo al procesador principal y liberar la memoria del dispositivo una vez finalizada la ejecución del kernel.

3.3.3. Invocaciones al Kernel

En una llamada a un kernel, se le ha de pasar el tamaño de grid y de bloque, por ejemplo, en el main del ejemplo anterior podríamos añadir:

```
1 dim3 bloque(N,N); //Definimos un bloque de hilos de N*N
2 dim3 grid(M,M) //Grid M*M
3
4 f<<<grid, bloque>>>(A, B, C);
```

Listing 3.4: Invocación al Kernel

En el momento que se invoque esta función, los bloques de un grid se enumerarán y distribuirán por los distintos multiprocesadores libres.

Cuando se invoca un kernel, el programador especifica el número de hilos por bloque y el número de bloques que conforman la malla. Una vez en la GPU, a cada hilo se le asigna un único número de identificación dentro de su bloque, y cada bloque recibe un identificador dentro de la malla. Esto permite que cada hilo decida sobre qué datos tiene que trabajar, lo que simplifica enormemente el direccionamiento de memoria cuando se trabaja con datos multidimensionales, como es el caso del procesado de imágenes o la resolución de ecuaciones diferenciales en dos y tres dimensiones.

Otro aspecto que destacar en la arquitectura CUDA es la presencia de una unidad de distribución de trabajo que se encarga de distribuir los bloques entre los Streaming Multiprocessors disponibles.

Los hilos dentro de cada bloque se ejecutan concurrentemente y cuando un bloque termina, la unidad de distribución lanza nuevos bloques sobre los Streaming Multiprocessors libres. Los Streaming Multiprocessors mapean cada hilo sobre un núcleo Streaming Processor, y cada hilo se ejecuta de manera independiente con su propio contador de programa y registros de estado. Dado que cada hilo tiene asignados sus propios registros, no existe penalización por los cambios de contexto, pero en cambio sí existe un límite en el número máximo de hilos activos debido a que cada Streaming Multiprocessor tiene un número determinado de registros.

3.3.4. Lanzamiento de un Kernel

En el siguiente ejemplo se van a definir dos funciones para efectuar la suma de dos números. Una de ellas se va a ejecutar en la CPU y devolverá el resultado a través de una sentencia return, por lo que se va a declarar como host y de tipo int, mientras que la otra se va a ejecutar en la en la GPU (el kernel) y por tanto se declarará como global y de tipo void. Además, la llamada al kernel se realizará utilizando la sintaxis “«<1,1>»”, lo que indica que se va a utilizar un solo bloque y un solo hilo:

```

1
2
3 // llamada a la funcion suma_GPU
4 suma_GPU<<<1,1>>>(m1, m2, dev_c);
5 // recogida de datos desde el device
6 cudaMemcpy( hst_c, dev_c, sizeof(int), cudaMemcpyDeviceToHost );

```

Listing 3.5: Lanzamiento de un Kernel

3.4. Hilos en CUDA

Una característica particular de la arquitectura CUDA es la agrupación de los hilos en grupos de 32, esta característica permite que un grupo de 32 hilos recibe el nombre de warp, y se puede considerar como la unidad de ejecución en paralelo, ya que todos los hilos de un mismo warp se ejecutan físicamente en paralelo y por lo tanto comienzan en la misma instrucción, cuando se selecciona un bloque para su ejecución dentro de un Streaming Multiprocessor, el bloque se divide en warps, se selecciona uno que esté listo para ejecutarse y se emite la siguiente instrucción a todos los hilos que forman el warp. Dado que todos ellos ejecutan la misma instrucción al unísono, la máxima eficiencia se consigue cuando todos los hilos coinciden en su ruta de ejecución.

Aunque el programador puede ignorar este comportamiento, conviene tenerlo en cuenta si se pretende optimizar alguna aplicación.

En cuanto a la memoria, durante su ejecución los hilos pueden acceder a los datos desde diferentes espacios dentro de una jerarquía de memoria. Así, cada hilo tiene una zona privada de memoria local y cada bloque tiene una zona de memoria compartida visible por todos los hilos del mismo bloque, con un elevado ancho de banda y baja latencia.

Finalmente, todos los hilos tienen acceso a un mismo espacio de memoria global ubicada en un chip externo de memoria DRAM. Dado que esta memoria posee una latencia muy elevada, es una buena práctica copiar los datos que van a ser accedidos frecuentemente a la zona de memoria compartida.

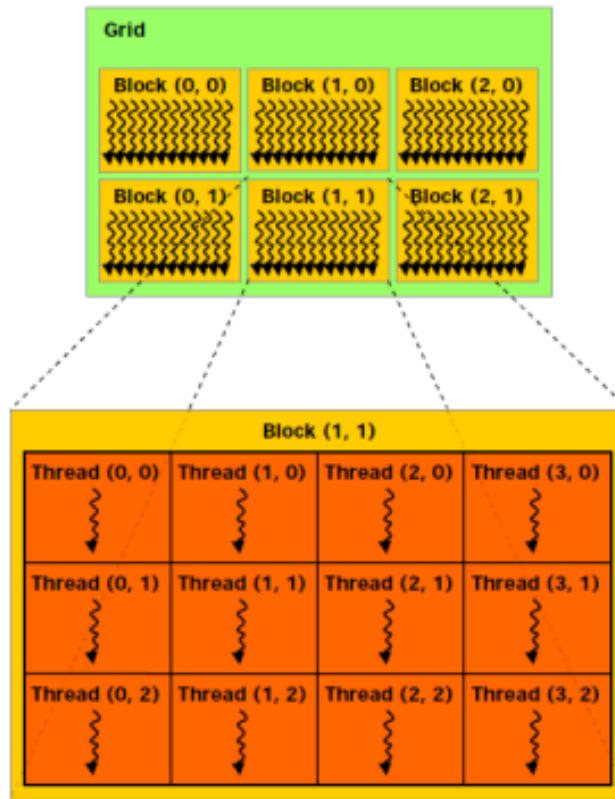


Figura 3.2: Jerarquía de Hilos

3.4.1. Paralelismo de los hilos

En CUDA es posible hacer uso del paralelismo a nivel de hilo que proporciona, lanzando más de un hilo. Las opciones para esta labor son múltiples, ya que se puede lanzar todos los hilos en un único bloque, o también lanzar varios bloques con un solo hilo cada uno, o la opción más flexible que sería lanzar varios bloques con varios hilos en cada bloque. Por ejemplo, para lanzar un kernel con N hilos de ejecución, todos ellos agrupados en un único bloque, la sintaxis sería:

```
1 myKernel<<<1,N>>>(arg_1,arg_2,...,arg_n);
```

Listing 3.6: Lanzamiento de un Kernel con N hilos

De este modo, la GPU ejecuta simultáneamente N copias de nuestro kernel. La forma de aprovechar este paralelismo que nos brinda la GPU es hacer que cada una de esas copias o hilos realice la misma operación pero con datos distintos, es decir, asignar a cada hilo sus propios datos.

Se puede identificar cada uno de los hilos para poder repartir el trabajo con una variable incorporada (built-in) denominada `threadIdx`, que es de tipo int y que únicamente puede utilizarse dentro del código del kernel. Esta variable adquiere un valor distinto para cada hilo en tiempo de ejecución. Cada hilo puede almacenar su número de identificación en una variable entera:

```
1 int myID = threadIdx.x;
```

Listing 3.7: Variable entera

Cuando se lanza el kernel mediante la sintaxis anterior, especificamos que queríamos una malla formada por un único bloque y N hilos paralelos. Esto le dice al sistema que queremos una malla unidimensional los valores escalares se interpretan como unidimensionales con los hilos repartidos a lo largo del eje x, ya que por defecto se considera este eje como la dimensión de trabajo.

De este modo, cada uno de los hilos tendrá un valor distinto de `threadIdx.x` que irá desde 0 hasta N-1. Es decir, cada hilo tendrá su propio valor de `myID` que permitirá al programador decidir sobre qué datos debe trabajar cada uno de ellos.

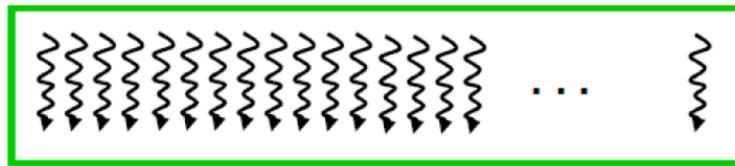


Figura 3.3: Malla formada por un único bloque de N hilos paralelos

El hardware de la GPU limita el número de hilos por bloque con que se puede lanzar un kernel. Este número puede ser mayor o menor dependiendo de la capacidad de cómputo de nuestra GPU y en particular no puede superar el valor dado por maxThreadsPerBlock, que es uno de los campos que forman parte de la estructura de propiedades cudaDeviceProp. Para las arquitecturas con capacidad de cómputo 1.0 este límite es de 512 hilos por bloque. Otra alternativa para lanzar un kernel de N hilos consistiría en lanzar N bloques pero de un hilo cada uno. En este caso la sintaxis para el lanzamiento sería:

```
1 myKernel<<<N,1>>>(arg_1,arg_2,...,arg_n);
```

Listing 3.8: Variable entera

Mediante esta llamada se tendría una malla formada por N bloques repartidos a lo largo del eje x y con un hilo cada uno

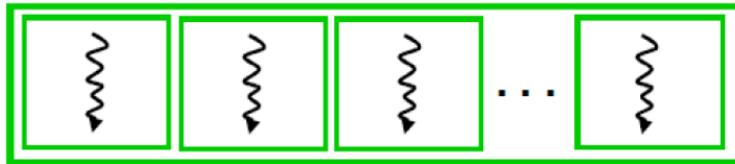


Figura 3.4: Malla formada por N bloques paralelos de 1 hilo repartidos a lo largo del eje x

El caso más general es el lanzamiento de un kernel con M bloques y N hilos por bloque en el cual se puede tener un total de $M \times N$ hilos. La sintaxis en este caso sería:

```
1 myKernel<<<M,N>>>(arg_1,arg_2,...,arg_n);
```

Listing 3.9: Variable entera

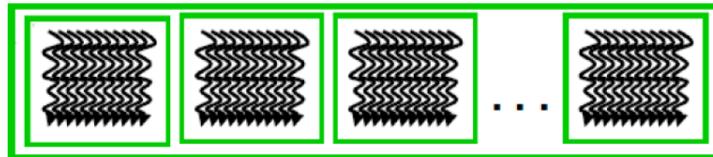


Figura 3.5: Malla formada por M bloques paralelos de N hilos cada uno

Ya que se tiene un total de $M \times N$ hilos ejecutándose en paralelo que para identificarlos será necesario hacer uso conjunto de las dos variables anteriores y de dos nuevas constantes que permitan a la GPU conocer en tiempo de ejecución las dimensiones del kernel que se han lanzado. Estas dos constantes son gridDim.x y blockDim.x. La primera nos da el número de bloques M y la segunda el número de hilos que tiene cada bloque. De este modo, dentro del kernel cada hilo se puede identificar de forma única mediante la siguiente expresión:

```
1 int myID = threadIdx.x + blockDim.x * blockIdx.x;
```

Listing 3.10: Variable entera

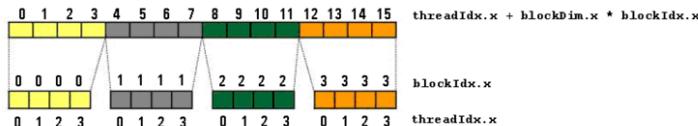


Figura 3.6: Identificación de los hilos dentro de una malla formada por cuatro bloques de cuatro hilos cada uno

3.4.2. Ejemplo

En este ejemplo se muestra forma de aprovechar el paralelismo de datos programando un kernel que realice la suma de dos vectores de longitud N inicializados con valores aleatorios comprendidos entre 0 y 1:

```

1 // GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
2 __global__ void suma( float *a, float *b, float *c )
3 {
4     int myID = threadIdx.x + blockDim.x * blockIdx.x;
5     // Solo trabajan N hilos
6     if (myID < N)
7     {
8         c[myID] = a[myID] + b[myID];
9     }
10 }
```

Listing 3.11: Declaración de funciones

```

1
2 int nBloques = N/BLOCK;
3 if (N%BLOCK != 0)
4 {
5     nBloques = nBloques + 1;
6 }
7 int hilosB = BLOCK;
8 printf("Vector de %d elementos\n", N);
9 printf("Lanzamiento con %d bloques (%d hilos)\n", nBloques, nBloques*hilosB);
10 suma<<< nBloques, hilosB >>>( dev_vector1, dev_vector2, dev_resultado );
```

Listing 3.12: Cálculo de bloques

El código completo se encuentra en el repositorio digital de GitHub: <https://github.com/FranciscoGuerrero58/IIMAS-Service>

3.5. Memoria en CUDA

La memoria global es la memoria de mayor tamaño que se puede encontrar en la GPU. A su vez, es la memoria con mayor latencia. Debido a su capacidad, esta memoria se utiliza normalmente como contenedor cuando copiamos datos entre el Host (CPU) y el Device (GPU). Posteriormente los datos pueden ser copiados a otras memorias más eficientes para optimizar el acceso a los datos, por ejemplo, a través del uso de la memoria compartida o registros.

La memoria global es compartida por todos los Streaming Multiprocessors de la GPU, por lo que todos los hilos podrán acceder al mismo espacio de direcciones de forma simultánea. El acceso a esta memoria tiene una elevada latencia, por lo que hay que minimizar su uso desde los kernels. Para intentar acelerar en la medida de lo posible el acceso a esta memoria, es muy recomendable seguir un patrón de acceso coalescente, de forma que hilos consecutivos accederán a posiciones contiguas en memoria. De esta forma, con la lectura de una línea de memoria, se podrá proporcionar datos para múltiples hilos de ejecución.

La manera correcta de reservar dinámicamente memoria global de la GPU es:

```

1 int *dev_a = 0;
2 // Reserva memoria en la GPU para un vector de enteros.
3 cudaStatus = cudaMalloc((void**)&dev_a, size * sizeof(int));
```

Listing 3.13: Reservar memoria global

3.5.1. Jerarquía de memoria en CUDA

El modelo de programación CUDA asume que tanto el host como el device mantienen sus propios espacios separados de memoria. La única zona de memoria accesible desde el host es la memoria global. Además, tanto la reserva o liberación de memoria global como la transferencia de datos entre el host y el device debe hacerse de forma explícita desde el host mediante llamadas a funciones específicas de CUDA.

Un kernel sólo pueden operar sobre la memoria del dispositivo, por lo que necesitaremos funciones específicas para reservar y liberar la memoria del dispositivo, así como funciones para la transferencia de datos entre la memoria del host y del device.

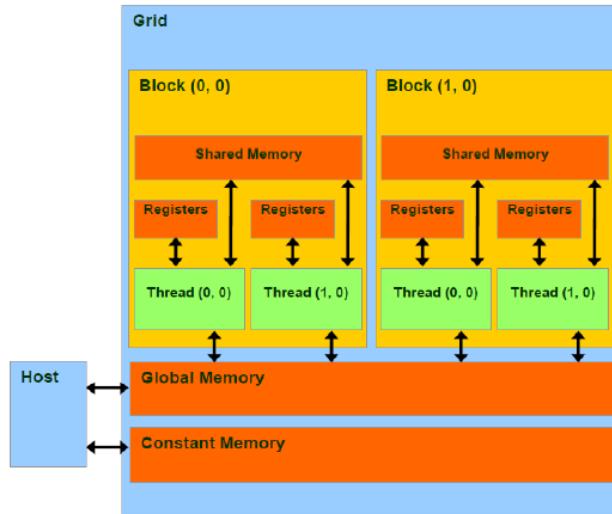


Figura 3.7: Jerarquía de memoria

Las únicas zonas de memoria accesibles desde el host son la memoria global (Global Memory) y la memoria constante (Constant Memory) y desde estas zonas de memoria el kernel puede transferir datos al resto de niveles. Se puede observar cómo todos los hilos pueden acceder a la zona de memoria global/constante, lo que resulta en un canal de comunicación entre todos ellos, mientras que únicamente sólo los hilos pertenecientes a un mismo bloque pueden acceder a una zona de memoria denominada memoria compartida.

Para poder reservar espacio en la zona de memoria global del device y poder acceder a ella desde el host se utiliza la función `cudaMalloc()`, que tiene un comportamiento similar a la correspondiente función estándar de C:

```
1 cudaMalloc(void **devPtr, size_t size);
```

Listing 3.14: Reservar memoria

El primer argumento de esta función es un doble puntero, que corresponde a la dirección del puntero (`devPtr`) en el que se va a almacenar la dirección de la memoria reservada en el dispositivo. El segundo argumento es la cantidad de memoria expresada en bytes que se desea reservar. De esta forma se puede reservar `size` bytes de memoria lineal dentro de la memoria global de la tarjeta gráfica.

Una vez que se tiene el espacio de memoria reservado en la memoria global del dispositivo, lo siguiente que se puede hacer es transferir datos entre esta memoria y la memoria de la CPU. Para ello se puede utilizar otra función parecida a la que se dispone en C estándar para tal efecto, sólo que en este caso se tiene algún parámetro adicional que permite especificar el origen y el destino de los datos. Esta función es `cudaMemcpy()`:

```
1 cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind);
```

Listing 3.15: Transferencia de datos

El primer parámetro (dst) corresponde al puntero con la dirección de destino de los datos, el segundo (src) es el puntero con la dirección de origen, es decir, donde se encuentran los datos que se desean copiar, count es el número de bytes que vamos a transferir y kind es el tipo de transferencia que se va a realizar.

3.5.2. Memoria compartida

Se sabe que cada hilo tiene sus propios registros y su propia memoria local. También que cada hilo tiene una memoria compartida que es visible para todos los hilos de su mismo bloque y que todos los hilos tienen acceso a la misma memoria global.

La característica más importante de la memoria compartida (shared memory) es que ésta se encuentra cerca de cada uno de los núcleos de procesamiento o que hace que sea una memoria con muy baja latencia. Debido a que está en el mismo chip que el procesador, la memoria compartida es mucho más rápida que los espacios de memoria local y global. Por ello, cualquier oportunidad de reemplazar los accesos a memoria global por accesos a memoria compartida debe ser aprovechada al máximo. En este sentido, lo más habitual será utilizar esta zona de memoria para realizar cálculos intermedios, para colocar datos que son frecuentemente accedidos, o bien para leer o escribir resultados que otros hilos del mismo bloque necesitan para su ejecución.

La memoria compartida se divide en módulos de memoria de igual tamaño denominados “bancos”, los cuales pueden ser accedidos de forma simultánea. Esto quiere decir que cualquier acceso a n direcciones de memoria que correspondan a n bancos diferentes pueden ser atendidos de manera simultánea, resultando en un ancho de banda que es n veces el correspondiente a un solo módulo. El problema está cuando dos accesos coinciden con el mismo banco, en cuyo caso los accesos se realizan de forma secuencial y el ancho de banda se ve seriamente reducido.

El hecho de que sólo los hilos pertenecientes al mismo bloque puedan compartir este espacio de memoria es otro parámetro de decisión a la hora de lanzar un kernel y organizarlo en diferentes bloques e hilos, además de las conocidas limitaciones del hardware.

La sintaxis para reservar memoria en este espacio se realiza a través del identificador shared dentro del código de la función global. Esto crea un espacio de almacenamiento para cada uno de los bloques que hemos lanzado sobre la GPU, pero con la característica de que los hilos de un bloque no pueden ver ni modificar los datos de otros bloques, lo que constituye un excelente medio por el cual los hilos de un mismo bloque pueden comunicarse y colaborar en la realización de un cálculo.

3.5.3. Ejemplo memoria compartida

```

1 __shared__ float cache[threadsPerBlock];
2 int tid = threadIdx.x + blockIdx.x * blockDim.x;
3 int cacheIndex = threadIdx.x;
4 float temp = 0;
5 while (tid < N) {
6 temp += a[tid] * b[tid];
7 tid += blockDim.x * gridDim.x;
8 }
```

Listing 3.16: Memoria compartida

3.5.4. Memoria constante

[1] La memoria constante, como su propio nombre indica, se usa para albergar datos que no cambian durante el transcurso de ejecución de un kernel. La capacidad de memoria constante disponible depende de la capacidad de cómputo de la GPU, y su principal ventaja es que en algunas situaciones el uso de memoria constante en lugar de la memoria global pueda reducir considerablemente el ancho de banda de memoria requerido por la aplicación. Dado que no se puede modificar la memoria constante, no se puede utilizarla para dejar los resultados de un cálculo, sino sólo valores de entrada.

El mecanismo para declarar memoria constante es similar al utilizado para declarar memoria compartida. Basta con utilizar el identificador `constant` delante de la variable.

Sin embargo, la declaración de variables constantes debe estar fuera del cuerpo de cualquier función. Por ejemplo, se puede reservar espacio para un array de 32 elementos de tipo float escribiendo al comienzo de nuestro código:

```
1 __constant__ float vector[32];
```

Listing 3.17: Memoria Constante

Esta declaración reserva estáticamente espacio en la memoria constante. De este modo no es necesario utilizar la función `cudaMalloc()`, pero es necesario decidir en tiempo de compilación el tamaño de las variables constantes.

El hecho de declarar memoria como constante restringe su uso a “sólo-lectura”. En principio, esta desventaja se ve compensada con el hecho de que se trata de una memoria con un elevado ancho de banda. Una de las razones de este elevado ancho de banda es que la memoria constante tiene asociado un nivel de memoria cache que se encuentra dentro de cada Streaming Multiprocessors (SM), por lo que lecturas consecutivas sobre la misma dirección no implican ningún tráfico adicional con la memoria. Otra razón del elevado ancho de banda se debe a una particularidad de la arquitectura CUDA y es que las lecturas realizadas por un determinado hilo se copian a otros 15 hilos del mismo warp.

Lógicamente esto es un beneficio sólo si todos los hilos tienen que acceder a los mismos datos. De no ser así, en vez de ser un beneficio podría llegar a ser incluso contraproducente.

3.5.5. Ejemplo memoria constante

```
1 // CUDA constants
2 __constant__ float dev_A[N][N];
3 // declaracion de funciones
4 // GLOBAL: funcion llamada desde el host y ejecutada en el device (kernel)
5 __global__ void traspuesta( float *dev_B)
6 {
7 // kernel lanzado con un solo bloque y NxN hilos
8 int columna = threadIdx.x;
9 int fila = threadIdx.y;
10 int pos = columna + N*fila;
```

Listing 3.18: Traspuesta de una matriz con memoria constante

3.5.6. Matriz cuadrada

En el siguiente ejemplo se muestran las diferencias y las similitudes que existen a la hora de reservar memoria tanto en el host como en el device. En este ejemplo se reserva espacio para una matriz cuadrada de $N \times N$ elementos, se inicializa en el host con valores aleatorios (entre 0 y 9) de tipo float y después se transfieren los datos desde el host hasta el device:

```
1 // declaracion
2 float *hst_matriz;
3 float *dev_matriz;
4 // reserva en el host
5 hst_matriz = (float*)malloc( N*N*sizeof(float) );
6 // reserva en el device
7 cudaMalloc( (void**)&dev_matriz, N*N*sizeof(float) );
8 // inicializacion de datos
9 srand( (int)time(NULL) );
10 for (int i=0; i<N*N; i++)
11 {
12     hst_matriz[i] = (float)(rand() % 10 );
```

Listing 3.19: Matriz con CUDA

3.5.7. Sincronización

El gran potencial de cálculo que nos proporciona CUDA al dividir una tarea en cientos o miles de hilos se basa en la posibilidad de que todos los hilos se puedan ejecutar de manera independiente y simultánea actuando sobre sus propios datos. Sin embargo, dado que los hilos son independientes, si esperamos que los hilos de un mismo bloque puedan cooperar compartiendo datos a través de alguna zona de memoria, también necesitamos algún mecanismo de sincronización entre ellos, es decir, necesitamos sincronizar su ejecución para coordinar los accesos a memoria. Por ejemplo, si un hilo A escribe un valor en una zona de memoria compartida y queremos que otro hilo B haga algo con ese valor, no podemos hacer que el hilo B comience su trabajo hasta que no sepamos que la escritura del hilo A ha terminado. Sin una sincronización, tenemos un riesgo de datos del tipo RAW (Read After Write) donde la corrección del resultado depende de aspectos no deterministas del hardware.

Para evitar este problema, en CUDA podemos especificar puntos de sincronización dentro del código del kernel mediante llamadas a la función:

```
1 syncthreads();
```

Listing 3.20: Función de sincronización.

Esta función actúa como barrera en la que todos los hilos de un mismo bloque deben esperar antes de poder continuar con su ejecución. Con esta llamada garantizamos que todos los hilos del bloque han completado las instrucciones precedentes a `syncthreads()` antes de que el hardware lance la siguiente instrucción para cualquier otro hilo. De este modo se puede saber que cuando un hilo ejecuta la primera instrucción posterior a `syncthreads()`, todos los demás hilos también han terminado de ejecutar sus instrucciones hasta ese punto.

Una aplicación donde se pone de manifiesto la necesidad de sincronizar la ejecución de los hilos es la conocida como “reducción paralela”. En general, un algoritmo de reducción es aquel donde el tamaño del vector de salida es más pequeño que el vector de entrada. Tenemos la suma de los componentes de un vector donde todos los hilos colaboran para realizar la suma. El punto clave estará en la sincronización de los hilos para que los resultados intermedios sean correctos.

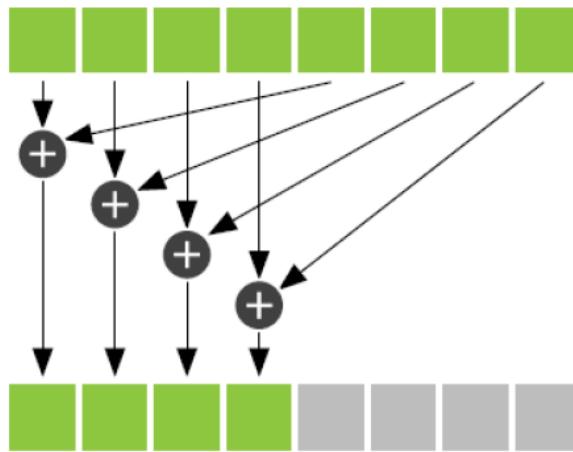


Figura 3.8: Jerarquía de Hilos

3.6. Arquitectura de CUDA

Es bien sabido que en la arquitectura clásica de una tarjeta gráfica se tienen dos tipos de procesadores, los procesadores de vértices y los procesadores fragmentados, cada uno de estos procesadores se dedica a tareas diferentes, cada uno cuenta con instrucciones diferentes y son independientes en un entorno gráfico. El hecho que los procesadores estén organizados de esta manera presenta dos problemas por un lado el desequilibrio de carga que aparece entre ambos procesadores y por otro la diferencia entre sus respectivos repertorios de instrucciones.

La arquitectura CUDA se encarga de estos problemas ya que todos sus núcleos de ejecución necesitan el mismo repertorio de instrucciones y prácticamente necesitan los mismos recursos.

Un multiprocesador contiene ocho procesadores escalares, dos unidades especiales para funciones trascendentales, una unidad multihilo de instrucciones y una memoria compartida. El multiprocesador crea y maneja los hilos sin ningún tipo de overhead por la planificación, lo cual unido a una rápida sincronización por barreras y una creación de hilos muy ligera, consigue que se pueda utilizar CUDA en problemas de muy baja granularidad, incluso asignando un hilo a un elemento por ejemplo de una imagen.



Figura 3.9: Jerarquía de memoria

3.6.1. ¿Con qué GPU's funciona CUDA?

CUDA funciona en todas las GPU de la marca NVIDIA de la serie G8X en adelante, incluyendo GeForce, Quadro, ION y la línea Tesla. La compatibilidad binaria es una de las grandes ventajas que se tiene ya que los programas que se han desarrollado en la serie GeForce 8 también funcionarán sin modificaciones en todas las futuras tarjetas NVIDIA, esta compatibilidad se debe por el conjunto de instrucciones PTX (Parallel Thread Execution).

Existe una gran variedad de tarjetas entre las que se encuentran:
Por parte de NVIDIA en los últimos años

- GeForce RTX 3090.
- GeForce RTX 3080.
- GeForce RTX 3070.
- GeForce RTX TITAN.
- GeForce 2080 SUPER.
- GeForce 2080 TI.

- GeForce 2080.
- GeForce 2070 SUPER.
- GeForce 2070.
- GeForce 2060 SUPER.

Se debe recordar que CUDA es una tecnología cerrada de NVIDIA, como consecuencia no es compatible con ATI que pertenece a AMD.

3.7. Aplicaciones

El procesamiento de imágenes es una herramienta de gran utilidad en diversas aplicaciones como vídeo vigilancia, reconstrucción de imágenes, información geográfica y médica. Sin embargo, estas aplicaciones requieren una gran demanda computacional para ser llevadas a cabo en el menor tiempo posible, aún y que se desarrollan nuevos algoritmos, suelen ser restrictivos para implementarse en tiempo real en sistemas que solo se basan en CPU. Afortunadamente, los algoritmos pueden ser analizados para llevarse a cabo en plataformas de cómputo paralelo, como las GPU-CUDA.

3.7.1. Imágenes Digitales

Hay situaciones donde puede ser interesante distribuir los hilos e ejecución de un kernel de CUDA de una manera análoga al problema que se pretende resolver, como son las operaciones con matrices.

Se puede definir una imagen “real” como una función continua bidimensional $f(x, y)$ que representa el valor de una intensidad, generalmente luminosa, en función de las coordenadas espaciales x e y . Cuando los valores de x , y y f son cantidades discretas y finitas, se denomina a la imagen como imagen “digital”. Por lo tanto, si se desea convertir una imagen real en una imagen digital lo que tenemos que hacer es convertir estas cantidades continuas en cantidades discretas. Para ello es necesario que se realicen dos procesos denominados muestreo y cuantificación.

El proceso de muestreo implica la discretización de las coordenadas espaciales x e y . El número de muestras tomadas de la imagen determina la resolución espacial, y a cada muestra se le denomina elemento de imagen o píxel.

El proceso de cuantificación implica la discretización de la amplitud de la señal f . El número de niveles de intensidad que se puede distinguir por de cada una de las muestras tomadas también denominados niveles de gris determina la resolución luminosa y depende del número de bits empleados.

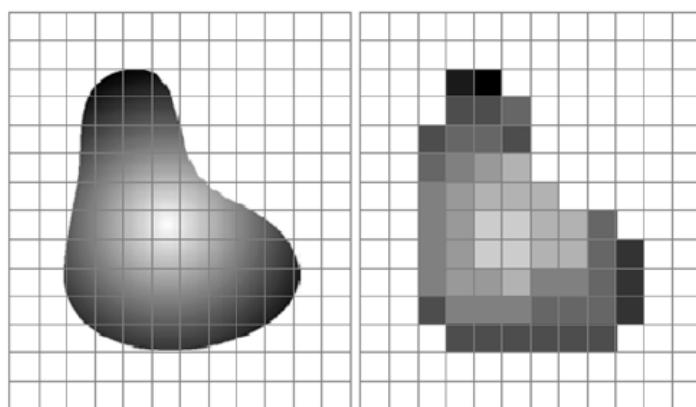


Figura 3.10: Ejemplo de Imagen digital que se obtuvo tras un proceso de muestreo y cuantificación.

Por ejemplo, si utilizamos k bits por cada muestra podremos representar cada una de ellas con 2^k niveles de gris distintos. Por tanto, utilizando 8 bits el número de niveles disponibles es 256. Vemos que tras el proceso de muestreo y cuantificación de una imagen lo que obtenemos es una matriz de números, es decir, que una imagen digital no es más que la representación de una imagen real a partir de una matriz numérica.

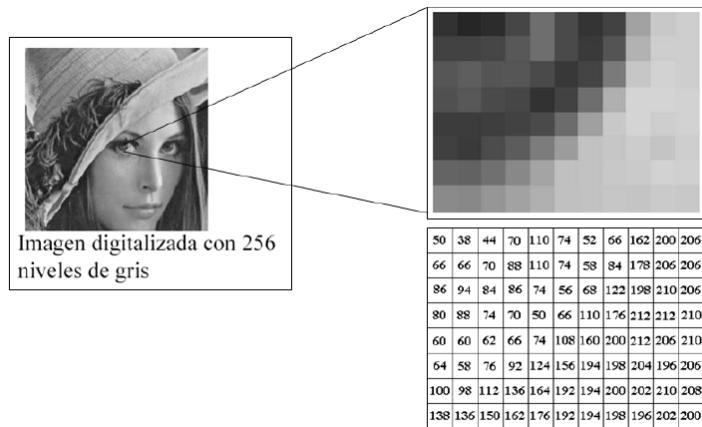


Figura 3.11: Imagen digital como una matriz de números.

3.7.2. Imágenes en color

Desde el punto de vista de la información almacenada en una imagen digital, se puede clasificar las imágenes digitales como:

- Imágenes de intensidad (escala de grises), donde cada elemento de la matriz representa un nivel de gris dentro del rango determinado por el número de bits empleado en la cuantificación
- Imágenes en color, donde para este tipo de imágenes se necesita tener información de tres colores primarios, definidos por el espacio de color que se esté utilizando.

El espacio de color que se utiliza para aplicaciones tales como la adquisición o generación de imágenes en color es el RGB, donde la combinación aditiva de los colores primarios rojo (Red), verde (Green) y azul (Blue) produce todo el rango de colores representables en dicho espacio.

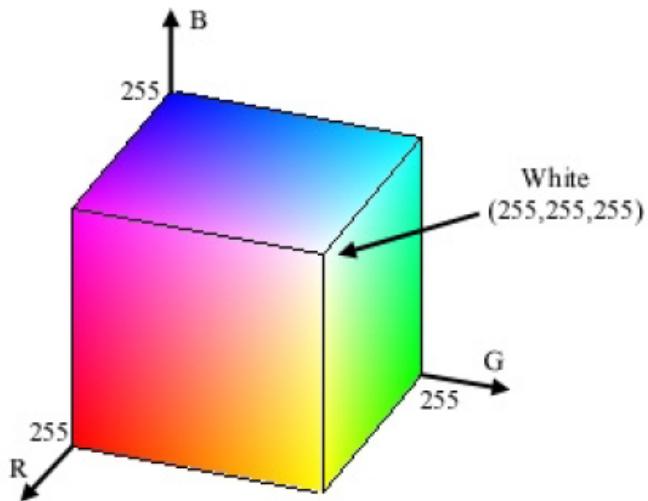


Figura 3.12: Representación del espacio de color RGB.

Una imagen en escala de grises también puede considerarse como una imagen en color en la que sus tres componentes son iguales. Así pues, para representar una imagen en color se necesitan como mínimo tres matrices correspondientes a las componentes roja, verde y azul de cada píxel individual.

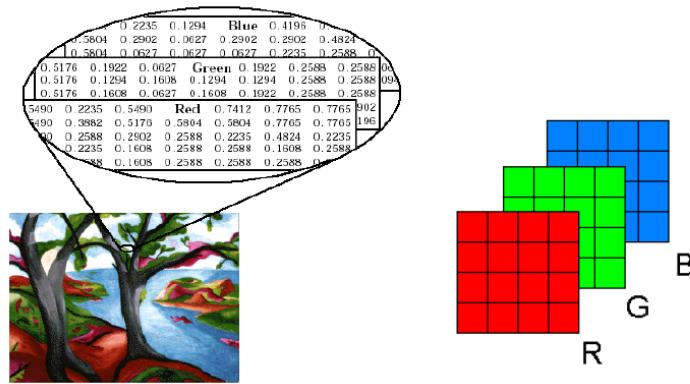


Figura 3.13: Imagen RGB.

3.7.3. Imágenes en mapa de bits

Un bitmap (o mapa de bits) es la forma natural de organizar en memoria la información de una imagen digital. Así, dado que una imagen digital se representa como una matriz rectangular de píxeles o puntos de color, en un mapa de bits la información de cada píxel se almacena en posiciones consecutivas de la memoria formando un array cuyo tamaño depende de la altura y la anchura de la imagen (número de píxeles) y de la información de color contenida en cada píxel (bits por píxel).

La información de color de cada píxel se codifica utilizando canales separados, cada uno de los cuales corresponde a uno de los colores primarios del espacio de color utilizado. A veces, se puede añadir otro canal que representa la transparencia del color respecto del fondo de la imagen y se denomina canal " α " (modelo RGBA). Por lo tanto, el tamaño necesario para almacenar en memoria un mapa de bits de una imagen en color de $M \times N$ píxeles, con 8 bits por canal y cuatro canales (R, G, B y α) es de $4 \times M \times N$ bytes. Y esta información se almacena en memoria principal ocupando posiciones consecutivas de memoria.

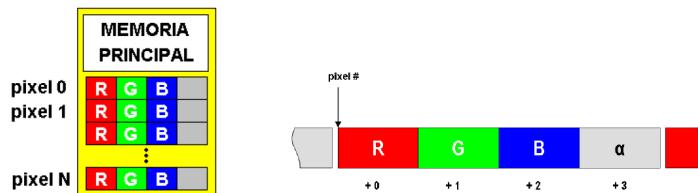


Figura 3.14: Memoria principal de un mapa de bits .

3.7.4. OpenGL

Con el fin de poder visualizar una imagen digital en el sistema se necesitan funciones que hagan la labor de abrir una ventana y de dibujar en ella. La forma más sencilla es utilizar una API Application Programming Interface - Interfaz de programación de aplicaciones destinada a tal efecto como lo es OpenGL.

OpenGL (Open Graphics Library) es una especificación estándar, es decir, un documento que describe un conjunto de funciones y el comportamiento exacto que deben tener. Consta de más de 250 funciones diferentes que pueden usarse para dibujar escenas tridimensionales complejas a partir de primitivas geométricas simples, tales como puntos, líneas y triángulos. El funcionamiento básico de OpenGL consiste en aceptar primitivas tales como puntos, líneas y polígonos, y convertirlas en píxeles. Dado que está basado en procedimientos de bajo nivel, requiere que el programador dicte los pasos exactos necesarios para renderizar una escena. Esto contrasta con otras interfaces descriptivas, donde un programador sólo debe describir la escena y puede dejar que la biblioteca controle los detalles para representarla. El diseño de bajo nivel de OpenGL requiere que los programadores conozcan en profundidad la pipeline gráfica, a cambio de darles libertad para implementar algoritmos gráficos novedosos.

El siguiente ejemplo muestra una forma muy sencilla de generar un bitmap y de mostrarlo por pantalla.

```

1 // generamos el bitmap
2 dim3 Nblocques(DIM/16,DIM/16);

```

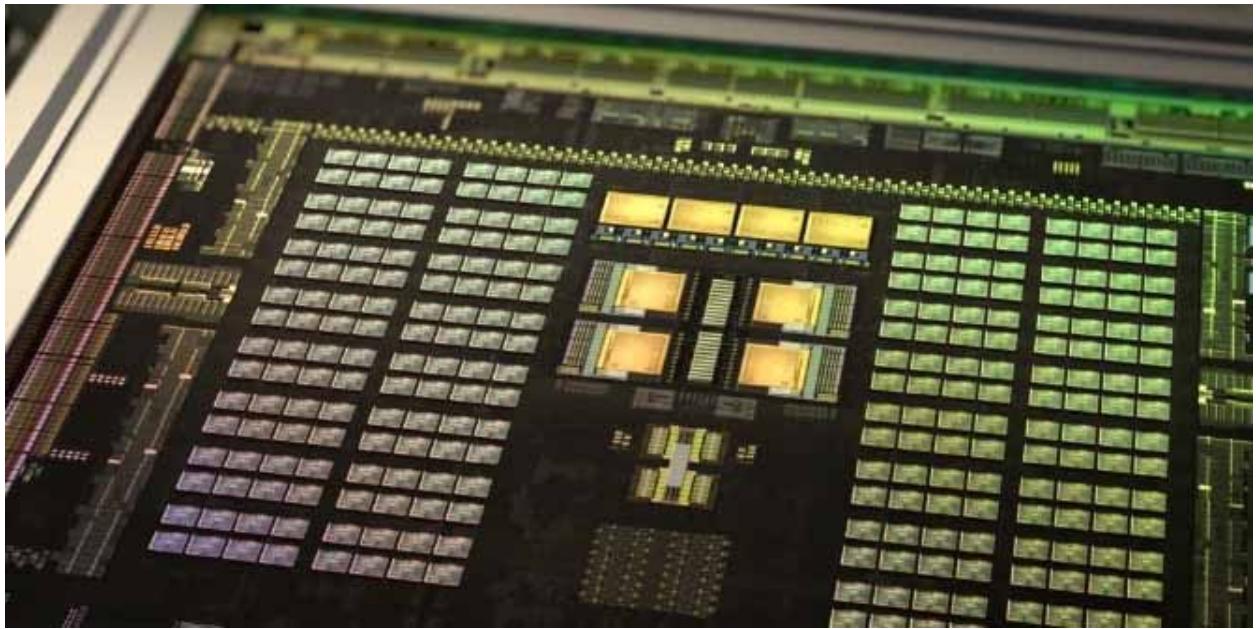
```
3 dim3 hilosB(16,16);
4 kernel<<<Nblocos,hilosB>>>( dev_bitmap );
5 // recogemos el bitmap desde la GPU para visualizarlo
6 cudaMemcpy( host_bitmap, dev_bitmap, size, cudaMemcpyDeviceToHost );
7 // liberacion de recursos
8 cudaFree( dev_bitmap );
9 //El codigo completo se encuentra en el repositorio digital: https://github.com/FranciscoGuerrero58/IIMAS-Service
```

Listing 3.21: Bitmap

El código anterior dibuja un bitmap donde cada hilo se encarga de dar color a un píxel, asignado a cada canal un valor relacionado con su posición espacial. El bitmap se dibuja en la pantalla de izquierda a derecha y de abajo a arriba, es decir, el píxel de coordenadas (0, 0) se sitúa en la esquina inferior izquierda.

Capítulo 4

CUDA con Julia



4.1. Introducción a CUDA.jl

Si bien se sabe que las GPU son aceleradores masivamente paralelos que tienen la capacidad de acelerar las aplicaciones de propósito general con uso intensivo de cómputo. Sin embargo, esa generalidad se encuentra restringida por la mayoría de las GPU, estas deben tratarse como un coprocesador lo que significa que utiliza espacios de memoria separados, controlados por un procesador host, en su mayoría son incapaces de realizar operaciones de entrada o salida, etc.

De esto resulta que las GPU son relativamente difíciles de programar, los programadores tienen que lidiar con las complejidades de la programación de coprocesadores y necesitan experiencia con la programación paralela para evaluar si se pueden resolver problemas específicos de manera efectiva en una GPU así al mismo tiempo, los entornos de desarrollo respaldados por proveedores para programar aceleradores de GPU generalmente funcionan con lenguajes de programación de bajo nivel, NVIDIA CUDA, utiliza CUDA C, mientras que las GPU de AMD e Intel dado que CUDA es una tecnología cerrada y exclusiva para NVIDIA, se programan con OpenGL.

Las construcciones en estos lenguajes de bajo nivel se corresponden estrechamente con las funciones de hardware disponibles, lo que permite alcanzar el máximo rendimiento, ya que se evitan abstracciones potencialmente costosas.

Sin embargo, la falta de tales abstracciones también complica la programación de GPU, ya que no solo requiere habilidades de programación paralela y conocimiento del dominio para mapear los problemas, sino

también competencia de programación de bajo nivel y conocimientos de hardware de GPU para las implementaciones reales.

Sabemos que CUDA se diseñó para trabajar con C o C++ los cuales son considerados lenguajes de alto rendimiento, pero que sucede cuando se vuelve un código difícil de entender y de mantener ya que se debe recordar que ambos también son lenguajes de bajo nivel, si se pone a comparar código entre C y C++ con un código de Julia, resultará más legible el código de Julia, ya que tiene un mayor soporte y se compila en las mismas instrucciones que la GPU de la versión de C++ y es considerablemente más rápido.

Por esta razón Julia que es un lenguaje de alto nivel tiene tantas ventajas, CUDA.jl es una biblioteca o paquete que permite el desarrollo de Julia con CUDA, este paquete es la entrada a programar GPU NVIDIA con Julia, esto lo realiza a través de varios niveles de abstracción, esto va desde arreglos que son muy fáciles hasta usar núcleos que usan API de bajo nivel, por su parte, algunas de las ventajas más significativas al trabajar con CUDA.jl son:

- No trabajar con punteros directos a la memoria de la GPU, en lugar de realizar esto CUDA.jl nos permite trabajar con **CuArray** que se comporta casi igual que un **Array** de tipo normal en la CPU.
- La asignación de memoria en GPU es mucho más rápida ya que en C++ tendríamos que asignar la memoria y luego copiar los datos de la CPU a la GPU nosotros mismos y en Julia usamos la función **cu** en una CPU Array y esta función maneja todo el proceso de transferencia por nosotros devolviendo un archivo **CuArray**.
- Con Julia el Kernel se compila dinámicamente en función de los tipos de entrada, lo que significa que el código puede funcionar sin importar que tipo de dato sea.

4.2. Hola Mundo en CUDA.jl

La instalación si es la primera vez que trabajas con CUDA deberás de revisar si tu computadora tiene una GPU compatible, una vez que verifiques que sea compatible deberás usar el administrador de paquetes de Julia para descargar el paquete de CUDA de la siguiente forma:

```
1 Pkg> add CUDA
```

Listing 4.1: Instalación

Y para probar el paquete deberás testearlo de la siguiente manera

```
1 Pkg> test CUDA
```

Listing 4.2: Instalación

4.2.1. Solución de problemas

En el caso que se presente algún problema con la instalación de CUDA.jl uno de los problemas más comunes es que no se pudo encontrar una instalación CUDA adecuada, esto significa que CUDA.jl no pudo encontrar o proporcionar un kit de herramientas de CUDA. Se debe de volver a ejecutar con la JULIA DEBUG y la variable de entorno establecida en CUDA.

Si se encuentra con este error al deshabilitar los artefactos mediante la configuración JULIA CUDA USE BINARYBUILDER=false, se debe de asegurar de que CUDA.jl pueda detectar las piezas necesarias, por ejemplo, colocando los archivos binarios y bibliotecas de CUDA en ubicaciones detectables es decir, en PATH y en la ruta de búsqueda de la biblioteca. Además, el CUDA HOMEentorno se puede usar para indicar a CUDA.jl dónde está instalado el kit de herramientas de CUDA, pero eso solo ayudará si el contenido de ese directorio no se ha reorganizado.

Si se encuentra con un error desconocido 999, hay varios problemas conocidos que pueden estar causándolo:

- Una discrepancia entre el controlador CUDA y la biblioteca de controladores: en Linux, busque pistas en dmesg
- El controlador CUDA está en mal estado: esto puede suceder después de la reanudación. Se debe de intentar reiniciar .

Sin embargo, en general, es imposible decir cuál es el motivo del error, pero es probable que Julia no tenga la culpa. Asegúrese de que su configuración funcione por ejemplo, se debe intentar ejecutar nvidia-smi, un binario CUDA C, etc, y si todo se ve bien, presente un problema.

Una vez que se tiene la instalación completa podemos pasar a realizar el primer programa en CUDA.jl. Como en otros lenguajes de programación, Hola Mundo es el programa más sencillo que se puede realizar y en CUDA.jl esto no es la excepción, la sintaxis adecuada para llevar a cabo este programa es la siguiente:

```

1 #include <iostream>
2 __global__ void kernel(void){
3 }
4 int main(void){
5     kernel<<<1,1>>>();
6     printf("Hello world \n" );
7     return 0;
8 }
```

Listing 4.3: Hello World en CUDA.jl

4.3. Programación en CUDA.jl

El paquete CUDA.jl proporciona tres interfaces distintas, pero relacionadas, para la programación CUDA:

- el CuArraytipo: para programar con matrices
- Capacidades nativas de programación del núcleo: para escribir núcleos CUDA en Julia
- Envolturas de la API de CUDA: para interacciones de bajo nivel con las bibliotecas de CUDA.

Gran parte de la pila de programación de Julia CUDA se puede usar simplemente confiando en el CuArraytipo y usando patrones de programación independientes de la plataforma como broadcast otras abstracciones de matriz. Solo una vez que se encuentre con un cuello de botella en el rendimiento o con alguna funcionalidad faltante, es posible que deba escribir un kernel personalizado o usar las API de CUDA subyacentes.

4.3.1. CuArraytipo

El CuArraytipo es una parte esencial de la cadena de herramientas. Principalmente, se usa para administrar la memoria de la GPU y copiar datos desde y hacia la CPU:

```

1 a = CuArray{Int}(undef, 1024)
2
3 # essential memory operations, like copying, filling, reshaping, ...
4 b = copy(a)
5 fill!(b, 0)
6 @test b == CUDA.zeros(Int, 1024)
7
8 # automatic memory management
9 a = nothing
```

Listing 4.4: CuArray

Más allá de la gestión de la memoria, existe una amplia gama de operaciones de matriz para procesar sus datos. Esto incluye varias operaciones de orden superior que toman otro código como argumento, como map, reduce o broadcast. Con estos, es posible realizar operaciones similares a las de un kernel sin tener que escribir sus propios kernels de GPU:

```

1 a = CUDA.zeros(1024)
2 b = CUDA.ones(1024)
3 a.^2 .+ sin.(b)
```

Listing 4.5: CuArray

Cuando sea posible, estas operaciones se integran con las bibliotecas de proveedores existentes, como CUBLAS y CURAND. Por ejemplo, la multiplicación de matrices o la generación de números aleatorios se enviarán automáticamente a estas bibliotecas de alta calidad, si los tipos son compatibles, y de lo contrario recurrirán a implementaciones genéricas.

4.3.2. Programación del núcleo con @cuda

Si una operación no se puede expresar con la funcionalidad existente para CuArray, o si necesita exprimir hasta la última gota de rendimiento de su GPU, siempre puede escribir un kernel personalizado. Los núcleos son funciones que se ejecutan de forma masiva en paralelo y se inician utilizando la @cuda macro:

```

1 a = CUDA.zeros(1024)
2
3 function kernel(a)
4     i = threadIdx().x
5     a[i] += 1
6     return
7 end
8
9 @cuda threads=length(a) kernel(a)

```

Listing 4.6: Macro

Este tipo de núcleos le brindan toda la flexibilidad y el rendimiento que ofrece una GPU, dentro de un lenguaje familiar. Sin embargo, no todo Julia es compatible, generalmente no puede asignar memoria, la E/S no está permitida y el código mal escrito no se compilará. Como regla general, se debe mantener los núcleos simples y solo transfiera el código de forma incremental mientras verifica continuamente que todavía se compila y ejecuta como se espera.

4.3.3. Envolturas de la API de CUDA

Para un uso avanzado de CUDA, se puede usar los contenedores de la API del controlador en CUDA.jl. Las operaciones comunes incluyen la sincronización de la GPU, la inspección de sus propiedades, el inicio del generador de perfiles, etc. Estas operaciones son de bajo nivel, pero para su comodidad se envuelven con construcciones de alto nivel. Por ejemplo:

```

1 CUDA.@profile begin
2     # code that runs under the profiler
3 end
4
5 # or
6
7 for device in CUDA.devices()
8     @show capability(device)
9 end

```

Listing 4.7: API de CUDA

4.4. Abstracciones de orden superior

Para poder programar GPU con matrices se deben utilizar de manera constante las abstracciones de matriz de orden superior de Julia, este tipo de operaciones son las que toman el código de usuario como argumento y especializan la ejecución en él. Con estas funciones, a menudo puede evitar tener que escribir núcleos personalizados. Por ejemplo, para realizar operaciones sencillas con elementos, puede utilizar mapo broadcast de la siguiente manera:

```
[language=python, caption=CuArray]
julia> a = CuArray{Float32}(undef, (1,2));

julia> a .= 5
1×2 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 5.0  5.0

julia> map(sin, a)
1×2 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 -0.958924  -0.958924
```

Dentro de CUDA.jl existen diversas ayudas en el código que nos permiten realizar diversas acciones para una codificación más rápida o un entendimiento mayor del código, accumulate es una de estas ayudas la cual puede ser usada para retener valores intermedios como se ve en el siguiente ejemplo:

```
julia> a = CUDA.ones(2,3)
2×3 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 1.0  1.0  1.0
 1.0  1.0  1.0

julia> accumulate(+, a; dims=2)
2×3 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 1.0  2.0  3.0
 1.0  2.0  3.0
```

Al utilizar esta ayuda se debe tener cuidado con el operador de tipo flotante debido a que este debe ser asociativo ya que la operación se realiza en paralelo scan. `scan!` Eso $f(f(a,b)c)$ debe ser equivalente a $f(a,f(b,c))$. La acumulación con un operador no asociativo en un CuArray no producirá el mismo resultado que en un Array.

4.4.1. Construcción e inicialización

Esto debería ir en la sección de abstracciones de orden superior

El CuArray tipo tiene como objetivo implementar la AbstractArray interfaz y proporcionar implementaciones de métodos que se usan comúnmente cuando se trabaja con matrices. Eso significa que puede construir CuArrays de la misma manera que los Array objetos regulares:

```
julia> CuArray{Int}(undef, 2)
2-element CuArray{Int64, 1}:
 0
 0

julia> CuArray{Int}(undef, (1,2))
1×2 CuArray{Int64, 2}:
 0  0

julia> similar(ans)
1×2 CuArray{Int64, 2}:
 0  0
```

La copia de memoria hacia o desde la GPU también se puede expresar usando constructores o llamando a `copyto!`:

```
julia> a = CuArray([1,2])
2-element CuArray{Int64, 1, CUDA.Mem.DeviceBuffer}:
 1
 2

julia> b = Array(a)
2-element Vector{Int64}:
 1
 2

julia> copyto!(b, a)
2-element Vector{Int64}:
 1
 2
```

4.4.2. Álgebra lineal

La funcionalidad de álgebra lineal de CUDA de la biblioteca CUBLAS se expone mediante la implementación de métodos en la biblioteca estándar LinearAlgebra:

```
julia> # enable logging to demonstrate a CUBLAS kernel is used
       CUBLAS.cublasLoggerConfigure(1, 0, 1, C_NULL)

julia> CUDA.rand(2,2) * CUDA.rand(2,2)
I! cuBLAS (v10.2) function cublasStatus_t cublasSgemm_v2(cublasContext*, cublasOperation_t, cublasOperation_t,
2×2 CuArray{Float32, 2, CUDA.Mem.DeviceBuffer}:
 0.295727  0.479395
 0.624576  0.557361
```

Ciertas operaciones, como la multiplicación matriz-matriz anterior, también tienen un respaldo nativo escrito en Julia con el fin de trabajar con tipos que no son compatibles con CUBLAS:

```
julia> # enable logging to demonstrate no CUBLAS kernel is used
       CUBLAS.cublasLoggerConfigure(1, 0, 1, C_NULL)

julia> CUDA.rand(Int128, 2, 2) * CUDA.rand(Int128, 2, 2)
2×2 CuArray{Int128, 2, CUDA.Mem.DeviceBuffer}:
 -147256259324085278916026657445395486093  -62954140705285875940311066889684981211
 -154405209690443624360811355271386638733  -77891631198498491666867579047988353207
```

Las operaciones que existen en CUBLAS, pero que todavía no están cubiertas por construcciones de alto nivel en la biblioteca estándar de LinearAlgebra, se pueden acceder directamente desde el submódulo de CUBLAS. Se debe tomar en cuenta que no necesita llamar a los contenedores C directamente (p. ej cublasDdot.), ya que muchas operaciones también tienen contenedores de alto nivel disponibles (p. ej dot.):

```
julia> x = CUDA.rand(2)
2-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 0.74021935
 0.9209938

julia> y = CUDA.rand(2)
2-element CuArray{Float32, 1, CUDA.Mem.DeviceBuffer}:
 0.03902049
 0.9689629

julia> CUBLAS.dot(2, x, y)
0.92129254f0
```

```
julia> using LinearAlgebra
```

```
julia> dot(Array(x), Array(y))
0.92129254f0
```

4.4.3. Matrices dispersas

La funcionalidad de matriz dispersa de la biblioteca CUSPARSE está disponible principalmente a través de la funcionalidad del paquete SparseArrays aplicado a los CuSparseArrayobjetos:

```
julia> using SparseArrays

julia> x = sprand(10,0.2)
10-element SparseVector{Float64, Int64} with 4 stored entries:
 [3]  =  0.585812
 [4]  =  0.539289
 [7]  =  0.260036
 [8]  =  0.910047

julia> using CUDA.CUSPARSE

julia> d_x = CuSparseVector(x)
10-element CuSparseVector{Float64, Int32} with 4 stored entries:
```

```
[3] = 0.585812
[4] = 0.539289
[7] = 0.260036
[8] = 0.910047

julia> nonzeros(d_x)
4-element CuArray{Float64, 1, CUDA.Mem.DeviceBuffer}:
0.5858115517433242
0.5392892841426182
0.26003585026904785
0.910046541351011

julia> nnz(d_x)
4
```

Para arreglos 2-D se puede usar el CuSparseMatrixCSCy .CuSparseMatrixCSR

Se puede acceder a la funcionalidad no integrada directamente en el submódulo CUSPARSE nuevamente.

4.5. Gestión de la memoria

Un de los aspectos más cruciales de trabajar con una GPU es administrar los datos que contiene. El CuArray tipo es la interfaz principal para hacerlo, la creación de un CuArray asignará datos en la GPU, la copia de elementos en él se cargará y la conversión de nuevo a un Array descargará valores en la CPU:

```
# generate some data on the CPU
cpu = rand(Float32, 1024)

# allocate on the GPU
gpu = CuArray{Float32}(undef, 1024)

# copy from the CPU to the GPU
copyto!(gpu, cpu)

# download and verify
@test cpu == Array(gpu)
```

Una forma más corta de realizar estas operaciones es llamar al constructor de copias, es decir, CuArray(cpu). Cuando se trabaje con varios dispositivos, se debe tener cuidado con la memoria asignada, las asignaciones están vinculadas al dispositivo que estaba activo cuando solicitó la memoria y no se pueden usar con otro dispositivo. Eso significa que no puede asignar un CuArray, cambiar dispositivos y usar ese objeto. Se aplican restricciones similares a los objetos de la biblioteca, como los planes CUFFT.

Para evitar esta dificultad, puede utilizar una memoria unificada a la que se puede acceder desde todos los dispositivos. Estas API están disponibles a través de contenedores de alto nivel, pero los CuArray constructores aún no las exponen:

```
using CUDA

gpus = Int(length(devices()))

# generate CPU data
dims = (3,4,gpus)
a = round.(rand(Float32, dims) * 100)
b = round.(rand(Float32, dims) * 100)

# CuArray doesn't support unified memory yet,
# so allocate our own buffers
buf_a = Mem.alloc(Mem.Unified, sizeof(a))
d_a = unsafe_wrap(CuArray{Float32,3}, convert(CuPtr{Float32}, buf_a), dims)
```

```

finalizer(d_a) do _
    Mem.free(buf_a)
end
copyto!(d_a, a)

buf_b = Mem.alloc(Mem.Unified, sizeof(b))
d_b = unsafe_wrap(CuArray{Float32,3}, convert(CuPtr{Float32}, buf_b), dims)
finalizer(d_b) do _
    Mem.free(buf_b)
end
copyto!(d_b, b)

buf_c = Mem.alloc(Mem.Unified, sizeof(a))
d_c = unsafe_wrap(CuArray{Float32,3}, convert(CuPtr{Float32}, buf_c), dims)
finalizer(d_c) do _
    Mem.free(buf_c)
end

```

4.5.1. Preservación de tipo

En muchos casos, es posible que no desee convertir sus datos de entrada en un archivo CuArray. Por ejemplo, con los envoltorios de matriz, y se desea conservar ese tipo de envoltorio en la GPU y solo cargar los datos contenidos. El paquete Adapt.jl hace exactamente eso y contiene una lista de reglas sobre cómo desempaquetar y reconstruir tipos como envoltorios de matriz para que podamos conservar el tipo cuando, por ejemplo, cargamos datos en la GPU:

```

julia> cpu = Diagonal([1,2])      # wrapped data on the CPU
2×2 Diagonal{Int64,Array{Int64,1}}


julia> using Adapt

julia> gpu = adapt(CuArray, cpu) # upload to the GPU, keeping the wrapper intact
2×2 Diagonal{Int64,CuArray{Int64,1,Nothing}}

```

Dado que esta es una operación muy común, la cu función lo hace convenientemente en lugar de realizarla el programador:

```

julia> cu(cpu)
2×2 Diagonal{Float32,CuArray{Float32,1,Nothing}}

```

Se deben de tomar en cuenta varias consideraciones por ejemplo: la cu función es obstinada y convierte la entrada de la mayoría de los escalares de coma flotante en Float32. Esta suele ser una buena decisión, ya que Float64 y muchos otros tipos de escalares funcionan mal en la GPU. Si esto no es deseado, se debe utilizar adapt directamente.

La recolección de basura se refiere a las instancias de este CuArray tipo las gestiona el recolector de elementos no utilizados de Julia. Esto significa que se recopilarán una vez que sean inalcanzables, y la memoria que contiene se reutilizará o liberará. No es necesario administrar la memoria manualmente, solo asegúrese de que sus objetos no sean accesibles es decir, que no haya instancias ni referencias.

4.5.2. Grupo de memoria

Un grupo de memoria retendrá los objetos y almacenará en caché la memoria subyacente para acelerar futuras asignaciones. Como resultado, puede parecer que su GPU se está quedando sin memoria cuando no es así. Cuando la presión de la memoria es alta, el grupo liberará automáticamente los objetos almacenados en caché:

```

julia> CUDA.memory_status()          # initial state
Effective GPU memory usage: 16.12% (2.537 GiB/15.744 GiB)
Memory pool usage: 0 bytes (0 bytes reserved)

```

```
julia> a = CuArray{Int}(undef, 1024); # allocate 8KB
julia> CUDA.memory_status()
Effective GPU memory usage: 16.35% (2.575 GiB/15.744 GiB)
Memory pool usage: 8.000 KiB (32.000 MiB reserved)

julia> a = nothing; GC.gc(true)

julia> CUDA.memory_status()          # 8KB is now cached
Effective GPU memory usage: 16.34% (2.573 GiB/15.744 GiB)
Memory pool usage: 0 bytes (32.000 MiB reserved)
```

Si por algún motivo se necesita recuperar toda la memoria caché, se debe llamar a CUDA.reclaim():

```
julia> CUDA.reclaim()

julia> CUDA.memory_status()
Effective GPU memory usage: 16.17% (2.546 GiB/15.744 GiB)
Memory pool usage: 0 bytes (0 bytes reserved)
```

Se deben de tomar en cuenta varios aspectos en este punto ya que nunca debería ser necesario reclamar memoria manualmente antes de realizar cualquier operación de matriz de GPU de alto nivel, es decir, la funcionalidad que asigna debe llamar al grupo de memoria y liberar cualquier memoria almacenada en caché si es necesario. Es un error si esa operación se ejecuta en una situación de falta de memoria solo si no se recupera la memoria manualmente de antemano.

Si necesita deshabilitar el grupo de memoria, por ejemplo, debido a la incompatibilidad con ciertas API de CUDA, configure la variable JULIA CUDA MEMORY POOL de entorno none antes de importar CUDA.jl.

4.6. Hilos

En CUDA.jl se puede usar con tareas e hilos de Julia, lo que ofrece una manera conveniente de trabajar con varios dispositivos o de realizar cálculos independientes que pueden ejecutarse simultáneamente en la GPU.

Cada tarea de Julia obtiene su propio entorno de ejecución local de CUDA, con su propio flujo, identificadores de biblioteca y selección de dispositivos activos. Esto facilita el uso de una tarea por dispositivo o el uso de tareas para operaciones independientes que se pueden superponer. Al mismo tiempo, es importante tener cuidado al compartir datos entre tareas.

Por ejemplo, se puede tomar un cálculo costoso ficticio y se puede ejecutar a partir de dos tareas:

```
# an expensive computation
function compute(a, b)
    c = a * b           # library call
    broadcast!(sin, c, c) # Julia kernel
    c
end

function run(a, b)
    results = Vector{Any}(undef, 2)

    # computation
    @sync begin
        @async begin
            results[1] = Array(compute(a,b))
            nothing # JuliaLang/julia#40626
        end
        @async begin
            results[2] = Array(compute(a,b))
        end
    end
end
```

```

        nothing # JuliaLang/julia#40626
    end
end

# comparison
results[1] == results[2]
end

```

Se pueden usar construcciones familiares de Julia para crear dos tareas y volver a sincronizar después (@async @sync), mientras que la función ficticia compute demuestra tanto el uso de una biblioteca la multiplicación de matrices usa CUBLAS como un kernel nativo de Julia. A la función se le pasan tres matrices de GPU llenas de números aleatorios:

```

1 function main(N=1024)
2     a = CUDA.rand(N,N)
3     b = CUDA.rand(N,N)
4
5     # make sure this data can be used by other tasks!
6     synchronize()
7
8     run(a, b)
9 end

```

La main función ilustra cómo se debe de tener cuidado al compartir datos entre tareas, las operaciones de GPU normalmente se ejecutan de forma asíncrona, en cola en un flujo de ejecución, por lo que si cambiamos de tarea y, por lo tanto, cambiamos los flujos de ejecución, debemos synchronize() asegurarnos de que los datos estén realmente disponibles.

Si la aplicación necesita realizar muchas copias entre la CPU y la GPU, podría ser beneficioso "fijar" la memoria de la CPU para que las copias de memoria asíncronas sean posibles. Sin embargo, esta operación es costosa y solo debe usarse si puede preasignar y reutilizar los búferes de su CPU. Aplicado al ejemplo anterior:

```

1 function run(a, b)
2     results = Vector{Any}(undef, 2)
3
4     # pre-allocate and pin destination CPU memory
5     results[1] = Mem.pin(Array{eltype(a)}(undef, size(a)))
6     results[2] = Mem.pin(Array{eltype(a)}(undef, size(a)))
7
8     # computation
9     @sync begin
10        @async begin
11            copyto!(results[1], compute(a,b))
12            nothing # JuliaLang/julia#40626
13        end
14        @async begin
15            copyto!(results[2], compute(a,b))
16            nothing # JuliaLang/julia#40626
17        end
18    end
19
20    # comparison
21    results[1] == results[2]
22 end

```

Las copias de memoria en sí mismas no se pueden superponer, pero la primera copia se ejecutó mientras la GPU aún estaba activa con la segunda ronda de cálculos. Las copias se ejecutaron mucho más rápido, si la memoria no estuviera anclada, primero se tendría que almacenar en un búfer de CPU anclado.

4.6.1. Subprocesos múltiples

El uso de tareas se puede extender fácilmente a múltiples subprocesos con la funcionalidad de la biblioteca estándar de subprocesos:

```

1 function run(a, b)
2     results = Vector{Any}(undef, 2)
3
4     # computation
5     @sync begin

```

```

6     Threads.@spawn begin
7         results[1] = Array(compute(a,b))
8         nothing # JuliaLang/julia#40626
9     end
10    Threads.@spawn begin
11        results[2] = Array(compute(a,b))
12        nothing # JuliaLang/julia#40626
13    end
14 end
15
16 # comparison
17 results[1] == results[2]
18 end

```

Al usar la Threads.@spawnmacro, las tareas se programarán para ejecutarse en diferentes subprocessos de la CPU. Esto puede ser útil cuando está llamando a muchas operaciones que "bloquean." en CUDA, por ejemplo, copias de memoria hacia o desde la memoria no anclada. Sin embargo, por lo general, las operaciones que sincronizan la ejecución de la GPU incluida la llamada a synchronize, se implementan de manera que le devuelven el rendimiento al programador de Julia, para permitir la ejecución simultánea sin requerir el uso de diferentes subprocessos de la CPU.

Se debe tomar en cuenta que el uso de múltiples subprocessos con CUDA.jl es una adición reciente y aún puede haber errores o problemas de rendimiento.

4.7. Ejecución de un Kernel

Para iniciar la exposición de primeros pasos en CUDA.jl (mas que ejecución de un kernel) sugiero que inicie con lo que menciona el capítulo ABSTRACCIÓN DE ORDEN SUPERIOR es decir, mencionar los arreglos de tipo CUDA y como estos objetos tienen un grado de abstracción que al operar con ellos no se tiene que programar kernel alguno como en CUDA-C, “automáticamente” se ejecutan en la GPU. Una vez esto se da pie a la ejecución de un kernel en CUDA.jl, donde se tendrá que explicar cómo se manejan los índices con base en rejilla o grid, bloque e hilos

La facilidad que tienen las GPU de utilizar matrices facilita en gran medida el cálculo en la GPU, debajo de este proceso hay muchos otros procesos, uno de ellos el kernel, pero ¿Cómo es que se crea un Kernel en la GPU? El código que ejemplifica la creación de un kernel es:

```

1 function gpu_add1!(y, x)
2     for i = 1:length(y)
3         @inbounds y[i] += x[i]
4     end
5     return nothing
6 end
7
8 fill!(y_d, 2)
9 @cuda gpu_add1!(y_d, x_d)
10 @test all(Array(y_d) .== 3.0f0)

```

Listing 4.8: Creación del Kernel

Este código ejemplifica muchas partes importantes de CUDA.jl como lo son los CuArray y el lanzamiento de un kernel, una de las principales ventajas es que una vez compiladas las instrucciones, las siguientes invocaciones son mucho más rápidas. Cada vez que se emita la instrucción @cuda se compilará el kernel para su ejecución en la GPU.

4.8. Curiosidades de CUDA.jl

Gracias a las diversas actualizaciones es posible ahora utilizar las API de CUDA para construir gráficos computacionales al igual que hay un nuevo asignador de memoria lo cual representa que habrá un rendimiento muy mejorado.

El tipo CuArray ahora admite elementos de datos de unión isbits.

Y ahora es posible escribir un programa de GPU que se ejecute simultáneamente gracias a las operaciones asincrónicas las cuales son requisito indispensable para programas concurrentes.

Este tipo de operaciones no se bloquean y hacen posible que se ejecuten al mismo tiempo otras operaciones mientras están en espera a que se complete la operación asincrónica, esto para las GPU es demasiado bueno debido a que casi todas las operaciones son asincrónicas ya que se ejecutan en diferentes dispositivos.

Bibliografía

- [1] J. Sanders and E. Kandrot. Cuda by example. *Adisson Wesley*, 1, 2010.