T0. INTRODUCCION PROGRAMACION PARALELA

¿Qué es?

Es una forma de cómputo en donde se ejecutan varias instrucciones simultáneamente o en donde varios procesadores trabajan juntos para la resolución de alguna tarea. Con ello se consigue más capacidad de cómputo dividiendo y modificando el problema.

Formas de paralelismo

Segmentación: divide una unidad funcional en etapas independientes, intercalando registros para almacenar datos intermedios pasando los datos de una etapa a otra en tiempos de reloj.

Réplica: aumenta el número de operaciones por unidad de tiempo, varias unidades ejecutan una operación completa, así se tienen tantas instrucciones como unidades funcionales.

Hay paralelismo interno que queda oculto en la arquitectura del ordenador aumentando la velocidad sin disminuir prestaciones (segmentación funciones) y el explícito que si es visible al usuario.

Tipos de procesadores

Procesador vectorial: trabajan sobre datos homogéneos, organización de memoria en módulos, mejor rendimiento en instrucciones vectoriales y muchos datos para llenar.

Procesador escalar: mejor rendimiento con instrucciones vectoriales y réplica/segmentación, cuatro etapas con duración de submúltiplo de ciclo de reloj.

Procesador segmentado: cuatro etapas, después de la etapa inicial se ejecutan instrucciones por ciclo de reloj, mejor con segmentación.

Las 4 etapas son *búsqueda, decodificación, ejecución, almacenamiento.*

Procesador supersegmentado: cada etapa se divide en subetapas y se lanzan sin completar ciclos de reloj (1/2 o 1/4).

Procesador superescalar: varias instrucciones de forma simultánea.

Multithreading: varias procesos ligeros a la vez compartiendo procesador y recursos.

Procesadores VLIW: instrucciones más largas, varias operaciones por instrucción.

Clasificación de Hwang-Briggs

Establece una aproximación a las clases de computadores paralelos fijando 3 configuraciones que son; *computadores pipeline, computadores matriciales y sistemas multiprocesador*.

Pudiendo distinguir:

- Paralelismo temporal: varias operaciones en el mismo instante en una unidad funcional.
- Paralelismo espacial: síncronos o asíncronos.

TEMA 1. PROGRAMACION DE PROCESOS MASIVAMENTE PARALELOS

GPU vs CPU

GPU	CPU
+ Cores, - Cache, - Control	- Cores, + Cache, + Control
Hardware control más simple	Hardware de control más complejo
Hardware para cálculo	Más rendimiento
Más eficiente en energía	Más flexibilidad
Programación más restrictiva	Cara en términos de poder
Gran ancho de banda (Flops*)	Bajo ancho de banda
Alta latencia, muchos datos más lento	Baja latencia, pocos datos muy rápido
Gran cantidad de hilos, ligeros	Cantidad limitada de hilos
Conseguir una eficiencia de 80% en CPU malo, conseguir un 20% en GPU bueno	

*Flops -> Operaciones flotantes por segundo.

Componentes GPU: GPU, memoria video, RAMDAC, disipador, ventilador y alimentación.

Compilador CUDA

Compilador basado en LLVM.

Para la compilación se utiliza driver NVCC, que invoca todas las herramientas de compilación necesarias (cudacc, g++, cl…) y reescribe los kernels CUDA para aprovechar el paralelismo.

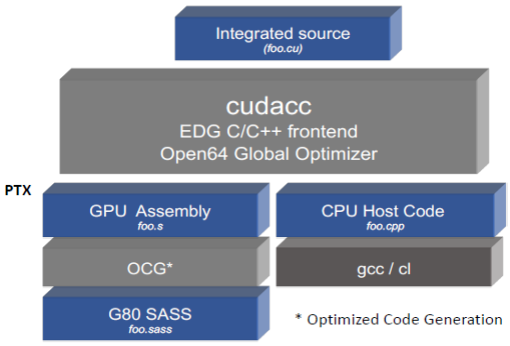
NVCC separa el código que se ejecuta en la CPU o en la GPU y produce:

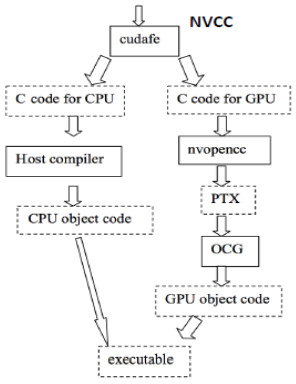
- Código en C: es para la CPU, se compila por separado con gcc, cl…
- Código PTX: código abierto que se interpreta en tiempo de ejecución.

Para un .exe en CUDA necesitamos dos librerías: CUDA runtime (cudart) y CUDA core (cuda).

Mediante nvcc -deviceemu se puede ejecutar un ejecutable desde el host, en donde cada hilo es un hilo del host. Se puede hacer depuración, acceder a un dato o llamar a una función de device a host o viceversa y detectar situaciones de deadlock.

Puede haber resultados diferentes ya que, mediante emulación los hilos se ejecutan secuencialmente y no simultáneamente, puede haber errores de punteros más fácilmente y además los resultados de cálculos en coma flotante será ligeramente diferentes.





Taxonomía de Flynn

Esta taxonomía se basa en el número de instrucciones concurrentes y de flujo de datos:

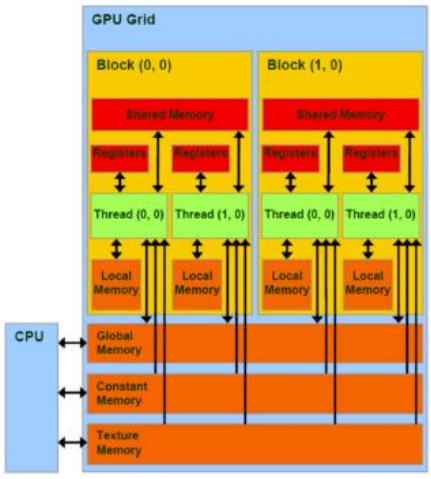
- SISD: Single Instruction Single DataStream (arquitectura x86)
- SIMD: Single Instruction Multiple DataStream (arquitecturas vectoriales)
- MISMD: Multiple Instruction Single DataStream (sistemas con tolerancia a fallos)
- MIMD: Multiple Instruction Multiple DataStream (arquitecturas distribuidas, multicore)

Las GPUs en la actualidad serían como un subtipo de las SIMD ya que desde un único programa se puede operar y acceder a flujos de datos de forma concurrente gracias a los hilos.

Jerarquía de memoria en CUDA

Ordenadas de mayor a menor latencia, sabiendo que en la mayoría de casos cuanta más velocidad menos capacidad tendrán:

- Global Memory (aplicación): accesible en R/W por todos los hilos y la CPU, haciendo que sea posible la comunicación GPU-CPU. El patrón de acceso a memoria puede afectar al rendimiento. OFF-CHIP
- Local Memory (hilo): es parte de la memoria global, es privada para cada hilo (memoria virtual). Es usada por el compilador para gestionar o guardar variables. OFF-CHIP.
- Texture Memory: controlada por el programador. Similar a la CM y solo R para GPU. Está cacheada, pero es OFF-CHIP.
- Constant Memory: es parte de la memoria global y de solo lectura, puede ser cargada en la cache de la SM para acelerar las transferencias. CPU tiene R/W pero los hilos de GPU solo R. OFF-CHIP
- Shared Memory (SM bloque): es limitada (16KB), accesible en R/W por los hilos de un mismo SM. Comunica entre los hilos de un mismo bloque teniendo cada bloque un espacio de SM. Su tiempo de vida es igual que el de un bloque. ON-CHIP.
- Registros: mismo núcleo para todos los hilos, accesible en R/W de forma privada. Cada hilo tiene su propio conjunto de registros. Están gestionados por el compilador. ON-CHIP.



Memoria	Localización	Cache	Acceso	Ámbito	Existencia
Registros	On-chip	N/A	R/W	Un thread	Thread
Compartida	On-chip	N/A	R/W	Threads dentro de un mismo bloque	Bloque
Global	Off-chip	No	R/W	Todos los threads + Host	Aplicación
Constantes	Off-chip	Sí	R	Todos los threads + Host	Aplicación
Texturas	Off-chip	Sí	R	Todos los threads + Host	Aplicación

Hilos en CUDA

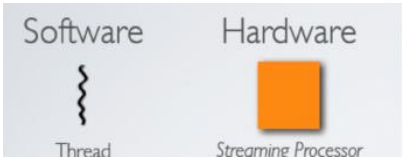
SP (Streaming Processor), SM (Streaming Multiprocessor).

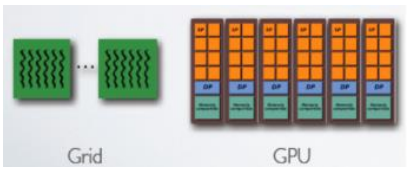
SPMD (Single Program Multiple Data).

Cada hilo en CUDA se ejecuta en un SP.

Los bloques de hilos se ejecutan en los SM y estos bloques no pueden ir de un SM a otro, por otro lado, la ejecución concurrente está limitada por el número de registros y capacidad de la memoria compartida.

Los kernels se ejecutan como grids de bloques habiendo un único kernel concurrente.





Un kernel ejecuta un array de hilos, teniendo todos los hilos el mismo código (SPMD), pero cada hilo tiene un ID diferente para acceder a memoria y poder controlar el comportamiento. Este array se divide en diferentes bloques, en donde los hilos de un bloque colaboran con memoria compartida, sincronización y operaciones, no pudiendo colaborar con los hilos de otro bloque.

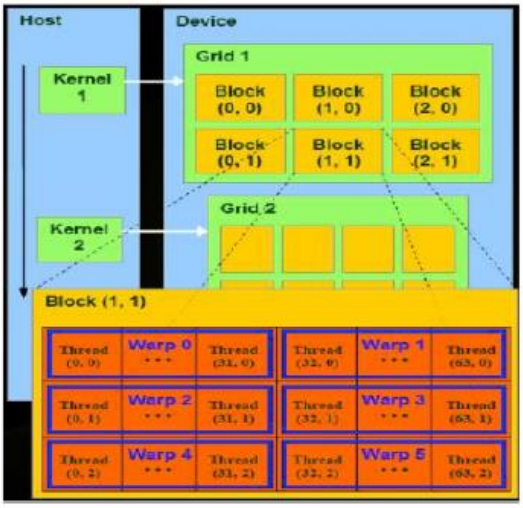
Que los hilos se ejecuten de esta manera permite que sea más fácil procesar datos que sean multidimensionales (matrices, procesamiento de imágenes…).

funcionKernel <<< tamañoGrid(bloques), tamañoBloque(hilos) >>>

Las llamadas a los kernels son asíncronas teniendo que utilizar herramientas de sincronización __syncthreads().

Funcionamiento y comportamiento de los hilos

Los hilos agrupados ya por bloques se asignan a un SM, teniendo estos SM un limite de bloques que pueden ejecutar. El SM mantiene el id hilo/bloque además de manejar y planificar su ejecución.



El hardware puede asignar bloques a cualquiera de los SM en cualquier momento.

A la hora de la ejecución de los hilos estos se agrupan en WARPS, los cuales son unidades de planificación en los SM.

Se planifica de la siguiente forma; los WARPS que están listos para ejecutarse pasan a ser elegible para su ejecución, pasando como a una “cola de espera” en donde se seleccionan para ejecutarse en función de una política de planificación priorizada, ejecutando todos los hilos de un WARP la misma instrucción.

Tiempos de acceso

Instrucciones

- Aritméticas -- 4 ciclos de reloj (suma, producto, MAD sobre coma flotante).

  (suma, operaciones nivel bit operandos enteros, max/min, conversión).

- Logarítmicas o cálculo de inversa – 16 ciclos de reloj.

- Multiplicación datos 32 bits – 16 ciclos de reloj.

Buena especialización en operaciones en coma flotante.

Memoria

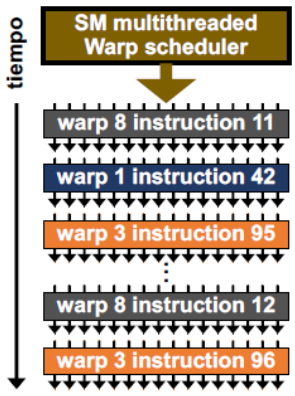
- Accesos – sobrecoste de 4 ciclos de reloj.

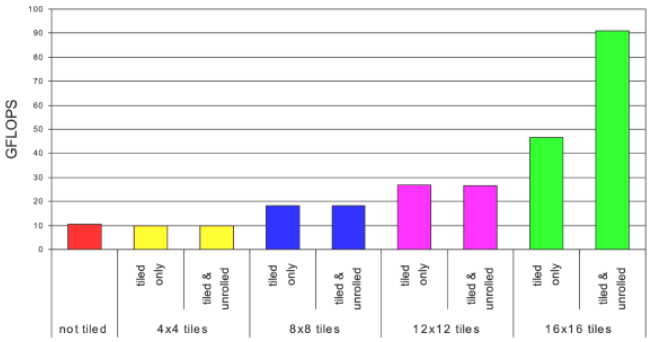
- Memoria global – 400-600 ciclos de reloj.

- Sincronización – sobrecoste de 4 ciclos de reloj + tiempo de espera.

Para optimizar el rendimiento se usa el algoritmo de baldosa, los cuales reducen los accesos a memorias mediante un factor llamado TILE_WIDTH agrupa los hilos de un bloque en teselas, haciendo que se aproveche al máximo el ancho de banda ya que no se necesitan hacer tantos accesos a memoria.

Los WARPS hacen que el tiempo de acceso a instrucción de un conjunto de hilos sea menor, ya que en vez de nHilos accesos se hace solo 1.





Compute capabilities

NVIDIA utiliza un formato para poder especificar las características de las diferentes tarjetas CUDA, a esto se le llama compute capabilities. Esto incluye dos números, el primero implica los cambios de generación y la segunda su revisión.

La primera fue compute capabilitiy 1.0 y las nuevas son compute capabilities 6.0 (Pascal) y la última versión 7.0 (Volta) llegando a la 7.5.

Consideraciones sobre hilos

Se lanza la cuadrícula en la SPA, los bloques se distribuyen a todos los SM que lanzan los WARPS de hilos, después los SM planifican y ejecutan los WARPS y una vez completados los recursos se liberan y SPA distribuye más bloques.

SPA	*Streaming Processor Array*: variable la serie GeForce 8 –la GeForce 8800 tiene 8-
TPC	*Texture Processor Cluster* (2 SM + TEX)
SM	*Streaming Multiprocessor* (8 SP) Núcleo multi-hilo Unidad de procesamiento para bloques de hilos en CUDA
SP	*Streaming Processor* ULA escalar para un único hilo en CUDA

Un hilo puede:

- R/W registros por hilo.
- R/W memorial local por hilo.
- R/W memoria compartida por bloque.
- R/W memoria global por grid.
- R memoria constante por grid.
- R memoria texturas por grid.

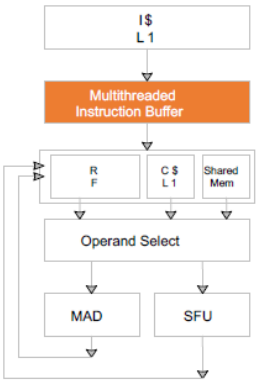
Consideraciones hardware

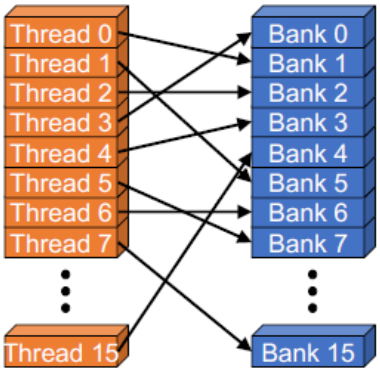
Para la planificación de WARPS se hace mediante el buffer de instrucciones el cual coge un instrucción WARP por ciclo de la cache de instrucciones a cualquier buffer, después ejecuta un ready to go WARP del buffer implementando la marcación de operandos (scoreboarding) y ejecuta un WARP basándose en Round-robin/edad y entonces el SM envía la misma instrucción al WARP seleccionado.

Los registros se reparten de forma dinámica y no automática (lo elige el programador) a lo largo de los bloques de un SM, si se asignan a un bloque no son accesibles por hilos de otro bloque.

Coalescencia: es la fusión de dos bloques libres adyacentes de memoria, para así hacer que todos los hilos de medio WARP accedan a direcciones contiguas.

En un máquina paralela muchos hilos acceden a memoria con lo que se **divide en bancos** para así poder servir una dirección de memoria por ciclo. Si dos hilos acceden al mismo banco habrá conflicto. Esto hace que varios hilos puedan acceder simultáneamente a la memoria. Podrán acceder tantos hilos a la vez como bancos haya. Si todos los hilos de medio WARP acceden a la misma posición no hay conflicto (broadcast).





Técnicas de optimización

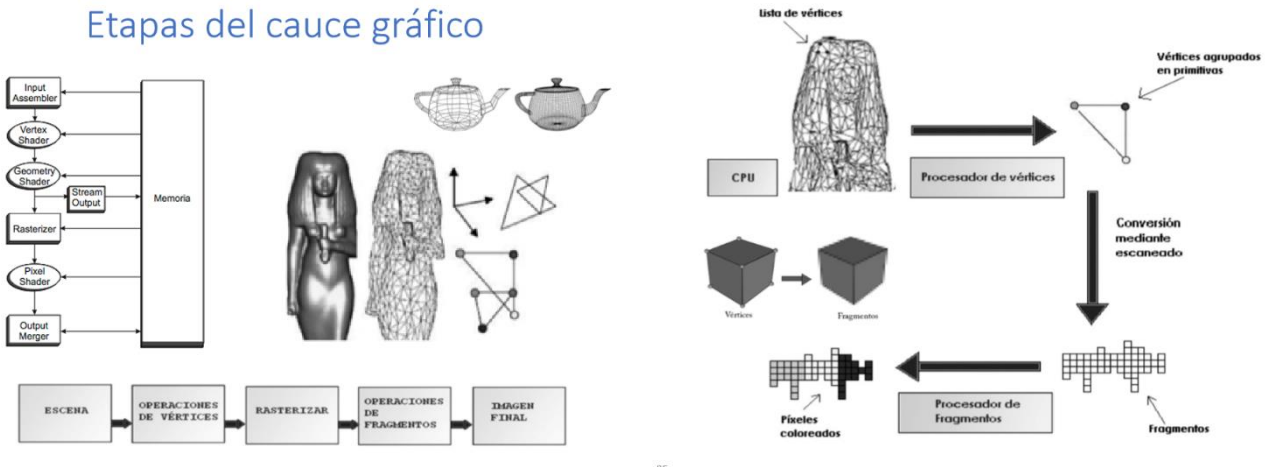
Unrolling: lo que hace es sustituir los bucles, que tienen condicionales y saltos por asignaciones de variables.

Cauce gráfico

Culling: comprueba que vértices de los triángulos están dentro o fuera del frustum, si todos los vértices están fuera se descartan, por el contrario, si hay vértices del triángulo dentro y fuera se pasa al clipping. Los triángulos se eliminan mediante back-face culling

Clipping: descarta parte de los triángulos que están fuera del frustum y los que están dentro y fuera se dividen.

Frustum: dentro de la imagen que se dibuja en un juego, es la parte visible en ese momento según la perspectiva.



Inconvenientes de la arquitectura

Algunos de los inconvenientes son:

- Programar vía API con un lenguaje que tengan que entender esas APIs.
- Tamaño de la memoria de textura es muy restrictiva, haciendo que sea limitada la capacidad de escritura.
- Capacidad de los shader, salida y acceso a memoria limitada
- Juego de instrucciones reducidos, faltan enteros y operaciones sobre bits. Se tenía que operar sobre matrices o vectores de 4 componentes
- Comunicaciones limitadas entre píxeles cuyas ubicaciones de memoria han sido predeterminadas, creando accesos raros.

G80 vs G200

G80	G200 (mejores)
Generación 1.0 - Tesla	Generación 1.3 – Tesla 2.0
128 cores CUDA	240 cores CUDA
512MB DDR3	1024MB
754 millones de transistores	1.400 millones de transistores
112 unidades de shader	240 unidades de shader
Menos rendimiento SFUs	Más rendimiento SFUs
TPCs agrupan 2 SM	TPCs agrupan 3 SM
Cada SM tiene 768 hilos	Cada SM 1024

SM en la arquitectura de Fermi tiene 32 SPs or 32 nucleos CUDA.

	CUDA Compute	
	1.0, 1.1	1.2, 1.3
Multiprocesadores / GPU	16	30
Cores / Multiprocesador	8	8
Hilos / Warp	32	32
Bloques / Multiprocesador	8	8
Hilos / Bloque	512	512
Hilos / Multiprocesador	768	1 024
Registros de 32 bits / Multiprocesador	8K	16K
Memoria compartida / Multiprocesador	16K	16K

Fermi vs Kepler

Fermi	Kepler
Generación 2.0 Nvidia	Generación 3.0 Nvidia
512 cores CUDA	1536 core CUDA
SMs 32 cores	SMs 192 cores
4 SFUs	4 SFUs
1536 hilos/SM	2048 hilos/SM
8 bloques/SM	16 bloques/SM
48 warps/SM	64 warps/SM
Más voltaje para funcionar	Menos voltaje para funcionar
Menos rendimiento/poder	Mayor rendimiento/poder

SFUs: Special Function Unit, para funciones especiales tales como seno y coseno.

Memoria en Fermi

Cada SM tiene dos planificadores WARPS y cada core tiene una unidad para el procesamiento de punto flotante y una ALU.

Memoria chip de 64KB como cache de primer nivel y otra parte como memoria compartida, alternando entre 16KB-48KB o 48KB-16KB.

Caché de segundo nivel de 768KB para acceso de memoria global.

Incorpora detección y corrección de errores para el acceso a datos.

T2. PROGRAMACIÓN FUNCIONAL

¿Qué es?

Paradigma de programación en el que se usan principalmente funciones puras y valores inmutables. En los lenguajes funcionales, todo es una expresión, es decir, todo tiene un valor inmutable.

Al usar variable inmutables nos aseguramos en todo momento que no se va a poder cambiar el valor de una variable y que las funciones nos devolverán un valor específico dependiendo de sus parámetros.

Pasos

Intérprete – shell interactiva, mismo tipos que Java, compilar(scalac) y ejecución(scala).

Variables – val(inmutable), var(mutable).

Funciones – funciones anónimas, asignación de un procedimiento a una variable.

Scripts – arrays empiezan en 0, array(index).

Estructuras de datos

Listas: Las más usadas, son inmutables

Método	Resultado
List() o Nil	Lista vacía
List("Uno", "Dos", "Tres")	Crea una lista con estos tres valores
val lista1 = "¡hola" :: "mundo" :: "!" :: Nil	Crea la lista lista1 con tres valores
List("Uno", "Dos"):::List("Tres")	Concatena las dos listas
lista1(2)	Devuelve "!"
lista1.head	Devuelve "¡hola"
lista1.tail	Devuelve una lista con "mundo" y "!"
lista1.reverse	Devuelve la lista en orden inverso
lista1.length	Devuelve el tamaño de la lista (3)
lista1.last	Devuelve "!"
lista1.isEmpty	Si la lista es vacía. Devuelve false

Tuplas: Como listas, pero agrupan objetos de tipos diferentes.

```
var tupla1 = (1, “uno”, List(1))
tupla1._1 //Primer elemento
tupla1._3
```

Conjuntos: Como listas y arrays pero puede trabajarse con mutables e inmutables, operación += ó +.

```
var conjunto = Set[Int]()
var conjunto = Set(1, 3, 5, 7, 11)
```

Mapas: Parejas de clave-valor, pueden ser mutables o inmutables, pueden tener tipos diferentes.

Método	Resultado
+=	Asigna/Agrega una nueva pareja al mapa
contains	Verifica si una llave está presente en un mapa
-=	Elimina una pareja del mapa especificando la llave de la misma
keys	Obtiene una colección iterable con las llaves del mapa
values	Obtiene una colección iterable con los valores del mapa

```
val m = Map(
   "hola" -> List('h', 'o', 'l', 'a'),
```

POO en Scala

En Scala las clases se definen y se instancian, los **atributos** de la **clase** se llaman **campos** y tienen **métodos** que definen su **comportamiento**. Podemos instanciar una clase y acceder a sus métodos y campos.

```
class Ave {
   var nombre = "aguila"
   def setNombre(n:String) =
      nombre = n
   def volar() =
      println("¡estoy volando!")
}
   var miAguila = new Ave
   miAguila.nombre --> String:ave
   miAguila.setNombre("Falcon")
   miAguila.nombre --> String:Falcon
```

Objetos Singleton

Métodos y valores que no están asociados con instancias individuales de una clase se denominan objetos singleton y se denotan con la palabra reservada **object** en vez de class. Se **utiliza para garantizar que una "clase" tenga una sola instancia**.

Se utilizan en Scala **para reemplazar los miembros de clase estáticos que existen en Java**.

```
object O {
   var nombre = "Singleton de O"
   def salida() = println(nombre)
}

O.Out
```

Muy similar a la definición de una clase (**object** por **class**)

Única instancia

Clase acompañante: se declara un **singleton** con el **mismo nombre** que una **clase**, el **singleton** es **companion object** y la **clase companion class**. Necesitan estar en el mismo archivo.

Singleton standalone: singleton **sin clases acompañantes**, se usa para **encapsular** y **agrupar** software, o para hacer una aplicación de Scala. Tienen **un método main** (como en la práctica).

```
object miPrograma {
   def main(args:Array[String]) {
      println("Mi aplicación")
   }
}
```

Definir método **Main**

Traits

Trait Application: tiene **declarado el método main**, por lo que, si extiende un objeto singleton, se puede hacer como aplicación Scala.

```
object Formas extends Application {
   val rec = new Rectangulo("rec1")
   rec.setCoords((5,5),(98,77))
   rec.print
}
```

Trait: encapsula definiciones de métodos y campos, que se pueden reutilizar mezclándolos dentro de clases. Son **como herencias**, pero se pueden usar en una misma clase varios traits, son como **nuevos tipos de variables**.

Para **herencia implícita** se usa **EXTENDS**, para **mezclar un trait que con una clase** que explícitamente extiende una superclase se usa **extends para superclase y WITH para trait**.

```
trait Rectangular {
   def coordSupIzq: (Int, Int)
   def coordInfDer: (Int, Int)
   def izq = coordSupIzq._1
   def der = coordInfDer._2
   def ancho = der - izq
}
```

```
class Rectangulo extends Rectangular {
   ...
}
```

Uso único de **extends**

```
class Rectangulo extends Figura with Rectangular {
   ...
}
```

Uso de **extends** y **with**

Trait doubling: crea un **trait que extiende a su vez otra trait** y que por tanto solo podrá ser usado en clases que **extiendan el trait superclase**.

```
import scala.collection.mutable.ListBuffer

abstract class IntQueue{
   def get(): Int
   def put(x: Int)
}

class BasicIntQueue extends IntQueue {
   private val buf = new ListBuffer[Int]
   def get() = buf.remove(0)
   def put(x:Int) = buf += x
}
```

```
trait Doubling extends IntQueue {
   abstract override def put(x:Int) = super.put(2 * x)
}
```

```
trait Incrementing extends IntQueue {
   abstract override def put(x:Int) = super.put(x + 1)
}
```

Definimos una cola que filtra números negativos

```
trait Filtering extends IntQueue {
   abstract override def put(x:Int) = if (x >= 0) super.put(x)
}
```

Añade uno a todos los números que almacena.

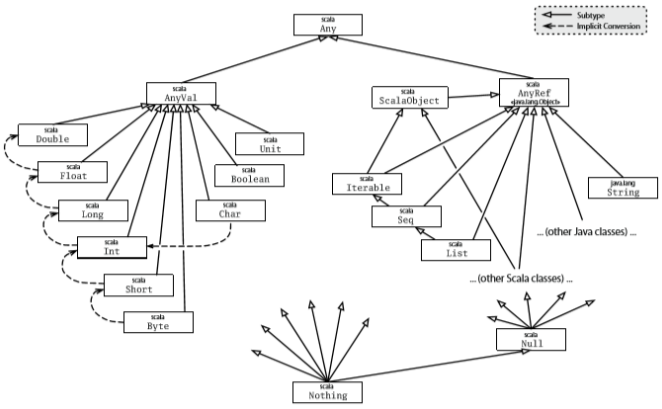
Mixins: son parecidos o los trait doubling, cuando se llama a un método de una clase con mixins, se llama primero al **método del trait más a la derecha** y van de **derecha a izquierda**, si este método a su vez se **llama super**, invocará **al método del siguiente trait a su izquierda**.

Jerarquía de clases

Jerarquía de clases en Scala

scala.Any -> superclase de todas las clases. Se divide en dos subclases:

- scala.AnyValclases para valores: se corresponden con tipos primitivos de los lenguajes tipo Java.
- Scala.AnyRefclases para referencias



Clase Any

- La clase Any define los siguiente métodos:
  - final def ==(that: Any): Boolean
  - final def !=(that: Any): Boolean
  - def equals(that: Any): Boolean
  - def hascode: Int
  - def toString: String

¿Se puede hacer override en las subclases de "==" y "!="?

¡¡Todo objeto en Scala puede usar estos métodos!!

Clase AnyVal

Define todos los **tipos de variables** en Scala, la mayoría son los **tipos primitivos** de los lenguajes de tipo Java, cada uno de sus **tipos tiene sus métodos** int(*, +, -), char(+)…

Clase AnyRef

La clase base de todas las **clase de referencia** en Scala, cuando se **define una clase por el usuario, esta extiende scala.AnyRef**. Toda **clase definida** por el **usuario extiende** de forma **implícita el trait scala.ScalaObject**

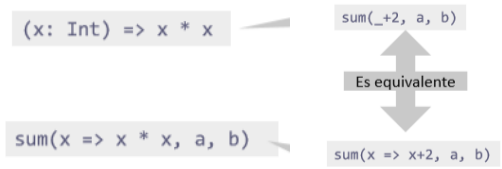
Funciones y closures

Funciones como objetos de primera clase

En Scala las funciones son también **objetos de primera clase**. Las funciones son de **primer orden**, cuando pueden **pasarse como argumento** o ser **devueltas como resultado** por otra función. Podemos:

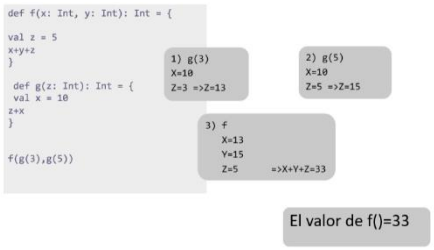
- Definir variables y parámetros de tipo función.
- Almacenar funciones en estructuras de datos como listas o arrays.
- Construir funciones en tiempo de ejecución (closures) y devolverlas como valor de retorno de otra función.

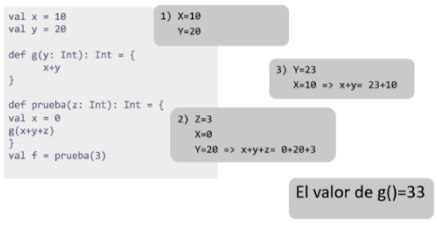
Funciones anónimas: son funciones **creadas en tiempo de ejecución**. Se pueden utilizar **placeholders**, que son huecos en los parámetros de la función.



Placeholders: se utilizan para asignar el **resultad de una función anónima a una variable.**

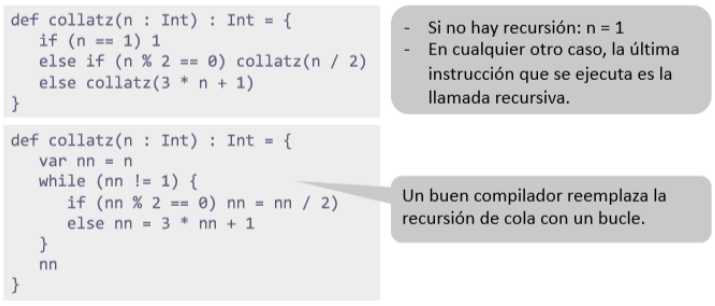
Clausuras: las funciones definidas en un ámbito **mantienen el acceso a ese ámbito** y pueden **usar los valores allí definidos**, es decir, se usan para asegurar que se **usan valores** de un **ámbito loca**l de una función. Llamamos clausura a estas funciones:





Tail recursion

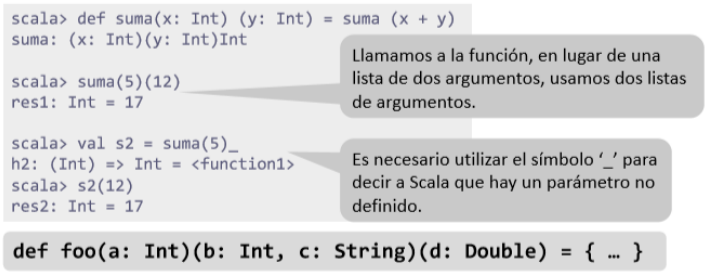
Una función es recursiva de cola si la **llamada recursiva es la última instrucción que se ejecuta**, en cualquier circunstancia, es decir, hacer una función recursiva hasta un caso base



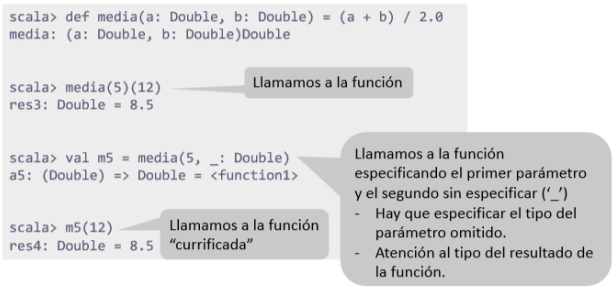
Currying

Es la técnica de **transformación una función que tiene múltiples argumentos** (tupla of argumentos) de tal manera que se le **puede llamar como una cadena de funciones cada uno como un único argumento** (aplicación parcial).

Transformar una función de varios argumentos en varias funciones de un argumento.




```
def foo(a: Int)(b: Int, c: String)(d: Double) = { … }
```



T3. CLOUD

¿Qué es?

Cloud computing está basado en la computación en Internet, donde los recursos se comparten, el software y la información son proporcionados a computadoras y otros dispositivos bajo demanda.

Tipos

On-deman computing: es cada vez más popular, se pone a disposición del usuario los recursos.

Ubiquitous computing: integración de la informática con las personas para que los ordenadores no se perciban como objetos diferentes.

Autonomic computing: nuevo paradigma, es un paradigma centrado en los datos.

Platform computing: soluciones y servicios de gestión de sistemas de baja latencia y alto rendimiento como IBM PlataformaComputing.

Edge computing: la totalidad o la mayor parte de la información de los datos se almacenan en la red.

Elastic computing: hace que se pueda aumentar o disminuir dinámicamente los recursos que se utilizan (memoria, procesamiento, almacenamiento) para satisfacer demandas.

Utility computing: suministran recursos computacionales tales como procesamiento, almacenamiento…, al igual que una compañía de luz por ejemplo.

Grid computing: utiliza de forma coordinada todo tipo de recursos.

Algo está cambiando…

- **Crecimiento exponencial** en las aplicaciones: biomedicina, exploración espacio, business analytics, web 2.0 social networking: YouTube, Facebook,…
- Generación de **contenidos escalable**: e-science and e-business data
- Gran **ratio de consumo de contenido digital**: Apple iPhone, iPad, Amazon Kindle,…
- Crecimiento exponencial en las **capacidades de computo**: multi-core, storage, bandwidth, virtual machines (virtualization)
- **Ciclos muy cortos de obsolescencia**: Windows Vista→ Windows 7; Java versions, Phython
- **Nuevas arquitecturas**: web services, modelos de persistencia, sistemas de ficheros distribuidos/repositorios (Google, Hadoop), multi-core, wireless,…
- No se puede manejar situaciones complejas con la infraestructura tradicional

Problemas existentes

Cuando Empresa contrata un desarrollo a medida necesita:

- **Servidor físico** dentro de la organización.
- Desplazamiento actualización de versiones
- **Servidor** dedicado para **datos**
- **Persona** que gestione el sistema, copias de seguridad, funcionamiento de los equipos, tener repuestos.
- **Conexión** a **internet** constante y buena.
- Si la app es más grande se tendría que **cambiar de servidor.**

Solución: Cloud Computing

- Requerimientos y modelos típicos:
  - software (SaaS),
  - platform (PaaS),
  - infrastructure (IaaS),

- Services-based application programming interface (API)
- Un entorno cloud provee uno o más de los requerimientos
- Suele facturarse en base al consumo
- Pueden ser públicas o privadas

SaaS (Software as a Service)

Engloba todas las **aplicaciones** que se pueden **usar a través de internet**, se conoce como **software a demanda**. Algunos ejemplos son Docs, Gmail, Netflix…

El **mantenimiento**, **actualización, copias de seguridad**, etc… corre a cargo del **proveedor** del servicio. EL **software** se **alquila** y no se compra, se realiza por medio de **subscripciones** o **pagos mensuales**.

Ventajas: no tienes costes de hardware, costes de alta, es escalable, actualizaciones automáticas, accesible desde cualquier lugar, son personalizables.

PaaS (Plataform as a Service)

Proporciona una **plataforma** y un entorno que permite a los **desarrolladores** crear **aplicaciones** y **servicio a través de internet** dándole todas las **herramientas** necesarias.

Se accede a través del **navegador web**, como por ejemplo Google App Engine, Azure, AWS.

Puede **incluir** SSOO, almacenamiento, soporte, hosting, herramientas de diseño y desarrollo.

Ventajas: no necesita inversion en cosas físicas, adaptabilidad, flexibilidad, colaboración de usuarios, seguridad…

IaaS (Infrastructure as a Service)

Proporciona acceso a **recursos informáticos situados en un entorno virtualizado**, espacio en servidores virtuales, conexiones de red, ancho de banda, direcciones IP... Algunos son Azure, AWS-EC2.

Se les dan los **medios** necesarios al **cliente** para **que pueda construir su plataforma infomática**.

Ventaja: tiene escalabilidad, no hace falta invertir en hardware, seguridad…

¿Nubes privadas, públicas o hibridas?

- nube privada están destinada a un uso exclusivo por parte de la empresa
  - requiere grandes medidas de seguridad tanto de los datos como de la plataforma en la que se almacenan
  - servicio de acceso y disponibilidad muy alto
  - Puede ser utilizado de forma interna (cloud privada interna) o por proveedores (cloud privada externa)
  - Los costes tanto de inversión como de mantenimiento suelen ser más altos que de otras nubes
- nube pública el servicio pertenece a un tercer proveedor y no, a la empresa
  - su uso no solo reside en la propia compañía sino también el suministrador del servicio cloud
  - infraestructura multi-uso, (diferentes usuarios o empresas)
  - forma gratuita o de pago
- Nube hibrida combinan soluciones privadas y públicas

Public Cloud vs. Private Cloud

Ventajas de las privadas:

- Seguridad y privacidad de los datos
- Lock-in del vendedor
- Altos requerimientos computacionales
- Reducción de costes al compartir la infraestructura entre los distintos proyectos de la empresa

Windows Azure

-Se ajusta a la **demanda**

-**Ciclos** y **almacenamiento** disponible **bajo solicitud a un coste**

-Se bene que usarla API de Azure para trabajar con la **infraestructura** ofrecida por **Microsoft**

-Las **características más significativas:** web role, worker role , blob1 storage, table y drive-storage

Amazon EC2

-Amazon EC2 es un **servicio web.**

- EC2 proporciona una **API para ejecutar instancias de cualquiera de los SSOO soportados**.

-**Facilita** la **computación** vía **Amazon Machine Images (AMIs)** para varios modelos.

-**Características:** S3, Cloud Management Console, MapReduce Cloud, Amazon Machine Image (AMI)

-**Ventajas**: Excelente distribución, balanceador de carga y herramientas de monitorizacion cloud.

Google App Engine

-Ofrece facilidades para el **diseño, desarrollo y desplegado de aplicaciones en Java** (o casi cualquiera soportado por la JVM), **Go and Python.**

- Ofrece las mismas características de servicio que en sus propias aplicaciones **(PAAS)**

-Interface está basado en la **programación**

-La escala resulta irrelevante (debido al modelo)

-**Características:** plantillas, excelente monitorización y gestión desde la consola

Aporta las siguientes **características a los desarrolladores:**

- Limita la responsabilidad del programador al desarrollo y primer despliegue. GAE provee recursos computacionales dinámicamente según son necesarios.

- Toma control de los picos de tráfico. Si nuestro portal crece en popularidad no es necesario actualizar nuestra infraestructura (servidores,BBDD). Ofrece replicación y balanceo de carga automática apoyado en componentes como Bigtable.

- Fácilmente integrable con otros servicios de Google. Los desarrolladores pueden hacer uso de componentes existentes y la librería de APIs de Google(email,autenticación,pagos,etc).

Características

-Ofrece una **plataforma completa** para el **alojamiento** y **escalado automático de aplicaciones,** consistiendo en: **Servidores** de aplicaciones **Python** y **Java**. **La base de datos BigTable**. El **sistema de ficheros Global file System(GFS)**

-Como desarrollador s**implemente tienes que subir tu código Python o Java compilado a Google**, **lanzar** la aplicación y **monitorizar** el uso y otras métricas.

-No todas las acciones se permiten (acceso a ficheros, llamadas al SO, algunas llamadas de red). Se **ejecuta en un entorno restringido** para permitir que las **aplicaciones escalen.**

GAE: Global File System

-**sistema de archivos de Google (GFS)**, es un **sistema de almacenamiento** basado en las **necesidades de Google**

se basa en las siguientes **premisas**:

-El sistema está construido para que el **fallo** de un componente **no le afecte.**

- El sistema almacena **grandes archivos**

-La mayoría del trabajo consiste en **dos tipos de lecturas**: **grandes** lecturas de datos y **pequeñas** lecturas aleatorias

- La carga de trabajo también consiste en añadir **grandes secuencias de datos a archivos.**

- El sistema debe ser **diseñado para ofrecer concurrencia a múltiples clientes** que quieran el **mismo archivo.**

- Tener un **gran ancho de banda** prolongadamente es más importante que una baja latencia.

GWT

-Esencialmente, es un **framework** para **crear aplicaciones AJAX** (Asynchronous JavaScript And XML) que usa tecnologías ampliamente extendidas.

-Las aplicaciones GWT pueden tener **varias ventanas contenidas bajo un único padre**.

-Es **sencillo** de usar (la construcción de UI es similar a la de Swing)

-**Reduce costes** (hay estimaciones que indican que se desarrolla 5x más rápido que con J2EE)

-**No se necesita servidor** y, si se usa, la mayoría de las computaciones se pueden delegar al cliente

- Se **optimiza el ancho de banda**

Manifiesto de GWT

-GWT **debe ayudar a crear código estable, eficiente y compatible con múltiples navegadores**

- GWT **debe ser amigable con los desarrolladores**. Compatible con IDEs, soportar depurado, refactorización, fuerte tipado,…