ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria
Corso di Laurea Magistrale in Ingegneria e Scienze Informatiche

TESTING AUTOMATICO PER UNA IMPLEMENTAZIONE DELLO STANDARD OPENGL

Elaborato in
SISTEMI AUTONOMI

Relatore
Prof. ANDREA OMICINI

Co-relatore
Dott. FABIO LAGALLA

Presentata da
EDOARDO ANTONINI

Seconda Sessione di Laurea
Anno Accademico 2017 – 2018

PAROLE CHIAVE

Testing

Automatizzazione

OpenGL

Amazon Elastic GPUs

Scripting

A mia nonna Mirella

Indice

Introduzione **ix**

1 Introduzione al cloud computing **1**
1.1 Premessa . . . 1
1.2 Modelli di servizio . . . 3
1.2.1 IaaS . . . 3
1.2.2 PaaS . . . 5
1.2.3 SaaS . . . 6
1.3 Attori del cloud computing . . . 6
1.4 Modelli di distribuzione . . . 7
1.5 Sicurezza nel cloud . . . 8
1.5.1 Modelli di tenancy . . . 10
1.6 Scelta di utilizzo del cloud . . . 10

2 Amazon Elastic GPUs **11**
2.1 Premessa . . . 11
2.1.1 Gestione di carichi di lavoro grafici . . . 11
2.1.2 APIs grafiche . . . 13
2.2 Scopo di Elastic GPUs . . . 17
2.2.1 Altre istanze grafiche in EC2 . . . 17
2.2.2 Vantaggi . . . 18
2.2.3 Limitazioni . . . 19
2.3 Struttura . . . 20
2.3.1 Utilizzare Elastic GPUs . . . 22

3 Metodologie di testing **23**
3.1 L'importanza del testing . . . 23
3.1.1 Testing nei vari modelli di sviluppo . . . 24
3.2 Test-Driven Development . . . 24
3.2.1 Code Coverage . . . 27
3.2.2 Test doubles . . . 27
3.3 Classificazione del testing . . . 28

3.3.1 Prospettive di testing 28
3.3.2 Tipi di testing 29
3.4 Code review vs Testing 34
3.4.1 Testare i test 35

4 Sviluppo del sistema di testing **37**
4.1 Premessa 37
4.2 Metodologia di lavoro 37
4.3 Situazione iniziale 38
4.3.1 Test pre-release 38
4.4 Tecnologie utilizzate 39
4.4.1 Linguaggi di scripting 39
4.4.2 Regex 40
4.5 Testare un implementazione di OpenGL - Compliance testing . 41
4.5.1 Khronos OpenGL Conformance Tests 41
4.5.2 Piglit 46
4.5.3 OpenGL Extensions Viewer 47
4.6 Regression test 48
4.6.1 Manual test 48
4.6.2 Image-based/rendering test 49
4.6.3 GUI-based testing 55
4.7 Performance testing 55
4.7.1 Benchmarking 56
4.8 OpenGL call recorder 57
4.9 Investigazione errori con EGPUs 58
4.9.1 Apitrace 59
4.10 Risultati finali 60

Conclusioni **61**

Ringraziamenti **63**

Bibliografia **65**

Sitografia **67**

Introduzione

In questi anni gli avanzamenti della tecnologia hanno portato ad un enorme diffusione dei dispositivi connessi alla rete, generando un incredibile aumento del traffico e dei dati. Il solo poter gestire ed analizzare questi dati può risultare molto profittevole per un'azienda, ma costruire in-house questo genere di servizi è incredibilmente costoso e difficile. Questo ed altri motivi hanno portato alla nascita del cloud computing, un modello per la condivisione on-demand di risorse computazionali. Tra i principali cloud provider va citato Amazon Web Services, uno dei leader di questo mercato e che di recente ha sviluppato una particolare tecnologia, Amazon Elastic GPUs.

Questo progetto nasce all'interno di Amazon EC2 e permette di collegare, in modo elastico, potenza grafica ad un'istanza che ne sarebbe sprovvista, in modo che l'utente possa scegliere la configurazione a lui più adatta e ottimizzare i costi.

Un servizio così complesso ha bisogno di essere sviluppato con le tecniche di ingegneria del software migliori, in modo da essere al meglio mantenibile, modulare e testabile. Quest'ultima caratteristica è importantissima in ogni fase del ciclo di vita del software, e diventa fondamentale durante la manutenzione, quando bisogna verificare che le modifiche non danneggino il comportamento originale del servizio.

Pur avendo un progetto con un buon livello di testabilità, è necessario utilizzare le corrette tecniche di testing per non impiegare troppe risorse durante le verifiche. Al giorno d'oggi si stanno diffondendo varie metodologie per effettuare il testing, ciascuna con pregi, difetti e scope diverso, molte delle quali hanno nel loro punto forte la possibilità di automatizzare, escludendo quasi completamente dal processo l'essere umano.

Nel mio progetto di tesi ho cercato di utilizzare queste metodologie per migliorare il grado di automazione di un'implementazione di OpenGL come Elastic GPUs, cercando anche di rendere i test più veloci possibile e ponendo particolare enfasi sull'espressività in caso di fallimento. Tutto ciò al fine di individuare eventuali errori in modo molto rapido e semplice.

Il documento sarà composto da quattro capitoli. Nel capitolo uno si darà un'introduzione alle tecnologie di cloud computing [1][2][9], definendo ad alto livello i differenti tipi di servizio ed accennando alle problematiche di sicurezza.

Nel secondo capitolo, invece, si entrerà più in dettaglio nell'implementazione di OpenGL [20][22][28] da testare, cioè Amazon Elastic GPUs [10][17][18][19], definendone obiettivi, vantaggi, limitazioni e struttura di riferimento.

Nella terza parte si darà una panoramica sulle metodologie di testing adottate al giorno d'oggi [3][4][5][6][11][12][13][14][16], iniziando dal motivo della loro importanza per poi arrivare ad un tipo di classificazione.

Il capitolo quattro tratterà del sistema di testing sviluppato e delle attività laterali relative alla suddetta tesi, definendo nel dettaglio tutti i tipi di test affrontati insieme agli altri compiti svolti, per poi presentare i risultati ottenuti.

Capitolo 1

Introduzione al cloud computing[1]

Con l'avanzare della tecnologia, la quantità e la potenza dei dispositivi in rete è incredibilmente aumentata. Gestire questi dispositivi e analizzare i dati che essi generano può essere molto profittevole e addirittura aumentare la qualità della vita media. Per essere effettivamente utili, i servizi che utilizzano queste informazioni devono avere alcune proprietà, tra cui, scalability, high avalability, reliability e security. Costruire, internamente ad un'azienda, applicazioni con queste caratteristiche può risultare molto difficile e costoso. Questo è stato uno dei principali motivi che hanno portato alla nascita del cosiddetto cloud computing, di cui parleremo in questo capitolo.

1.1 Premessa

Non esiste una definizione univoca di cloud computing, tuttavia, come punto di partenza, si può considerare la specifica del NIST[2] che dice:

"Il cloud computing è un modello per l'abilitazione di un pool condiviso di risorse computazionali configurabili (per esempio reti, server, supporti di archiviazione, applicazioni e servizi) in modo ubiquo, conveniente, on-demand e attraverso la rete, che può essere fornito e rilasciato in maniera veloce con un minimo sforzo di gestione o di interazione con il service provider" (Tradotto dall'inglese)

Il nome cloud computing deriva dal fatto che, molto spesso, viene utilizzato il simbolo della nuvola per nascondere la complessità dell'infrastruttura inter-

[1] Alcuni spunti sono stati tratti da: "Fondamenti di cloud computing" - slide del corso Smart City Luca Calderoni, Dario Maio

[2] National Institute of Standards and Technology

na della piattaforma e per rendere l'idea che i dati siano disponibili sempre, ovunque e in modo capillare.

La logica del cloud computing è quindi offrire delle risorse tecnologiche come un servizio, come succede per esempio, con l'acqua o l'elettricità, in modo che l'utente le possa utilizzare pur non possedendole.

Le risorse fornite possono essere sia fisiche che logiche, nel primo caso si può trattare di potenza di calcolo o memoria, mentre nel secondo di applicazioni, come delle web-mail.

Il successo del cloud computing è dovuto ad alcune caratteristiche particolari, varie già citate, che andremo a definire meglio di seguito:

- Accesso ubiquo tramite la rete
- Elasticità rapida
- Servizio misurato
- On-Demand Self-Service
- Resource Pooling

La prima caratteristica specifica come sia i dati che i servizi debbano essere accessibili continuamente tramite la rete. I client che li utilizzano potranno essere molteplici, e qui potranno sorgere problemi di controllo degli accessi.

Con "elasticità rapida" si intende il fatto di poter espandere o ridurre le risorse in uso, su richiesta e in modo molto rapido, viene considerata la "killer-feature" del cloud.

La terza proprietà si riferisce al fatto che la completa infrastruttura cloud è controllata con varie metriche dal cloud provider per poter ottimizzare l'utilizzo delle risorse.

L'On-Demand Self-Service è la capacità dell'utente di ottenere e modificare risorse senza fisica interazione con il service provider.

L'ultima caratteristica si riferisce al grande pool di risorse che viene utilizzato per rispondere alle esigenze dei clienti, queste ultime spesso sono condivise tra più utenti, qui è importante definire il livello di isolamento tra gli stessi come dettaglieremo in 1.5.1

1.2 Modelli di servizio

Un piattaforma cloud può offrire vari tipi di servizio, solitamente vengono divisi in tre categorie (figura 1.1[3]):

- IaaS (Infrastructure as a service)
- PaaS (Platform as a service)
- SaaS (Software as a service)

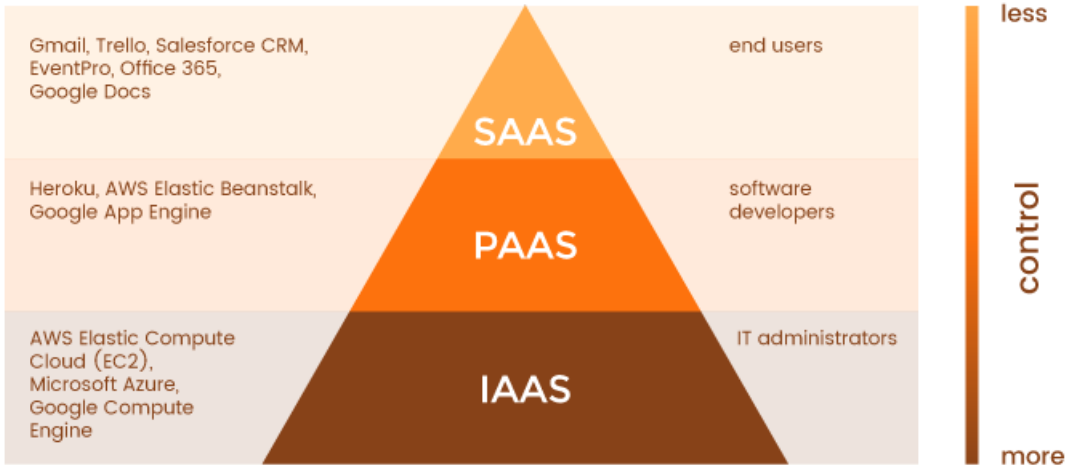

Figura 1.1: Modelli di servizio cloud

A volte vengono utilizzate anche altre terminologie più specifiche come per esempio GaaS (Gaming as a service) oppure SECaaS (Security as a service), tuttavia di seguito si vanno a definire più in dettaglio solo le tre categorie "tradizionali"

1.2.1 IaaS

I servizi IaaS sono quelli che si trovano al livello più basso della piattaforma cloud, essi permettono di manipolare direttamente risorse hardware, firmware e sistemi operativi.

L'utente in questo modo potrà scegliere quante macchine utilizzare per un certo servizio, specificando anche le caratteristiche dei nodi stessi e le connessioni di rete tra loro. La virtualizzazione, qui, è una caratteristica fondamentale,

[3]Da https://rubygarage.org/blog/iaas-vs-paas-vs-saas

infatti, fornendo all'utente un nodo virtuale invece di uno fisico, si può raffinare la granularità dell'utilizzo delle risorse permettendo di aumentare la flessibilità. Questo permette di fornire vari tipi di funzionalità, dal load balancing alla distribuzione e replicazione dei dati su nodi geograficamente sparsi.

Tra i tre modelli di servizio, IaaS è sicuramente quello più potente e flessibile, tuttavia, per poter utilizzarlo al meglio, è necessario disporre di personale con esperienza in questo settore, infatti spesso la configurazione di servizi di virtualizzazione hardware può risultare complessa.

AWS e Amazon EC2

Di seguito si fa una brevissima introduzione ad Amazon Web Services (AWS) e Amazon Elastic Compute Cloud (EC2) poiché sono stati largamente utilizzati durante il lavoro di tesi.

AWS è la piattaforma di cloud computing di Amazon, ed offre un gran numero di servizi per aiutare le crescita delle imprese e dei propri clienti. Le funzionalità offerte sono le seguenti:

- Compute
- Storage
- Database
- Migration
- Network and Content Delivery
- Management tools
- Security & Identity Compliance
- Messagging

Per il momento si entra nel dettaglio solo di Amazon EC2, un servizio IaaS del settore Compute utile alla gestione di server virtuali nel cloud.

In particolare, mette a disposizione una capacità computazionale elastica e sicura, con lo scopo di semplificare il lavoro degli sviluppatori e renderli capaci di scalare con facilità. In EC2 è possibile configurare le proprie macchine con velocità, in modo sicuro ed affidabile, sfruttando l'ecosistema Amazon che permette anche l'integrazione con vari altri servizi, come per esempio Amazon S3 (per lo storage) o Amazon VPC (Virtual Private Cloud). Una caratteristica importante di questa piattaforma è il pay-per-use, che garantisce che il cliente paghi solo per le risorse che usa e nient'altro. L'utente finale ha inoltre il

completo controllo delle macchine che utilizza, con un accesso root in ogni momento, in modo da avere completa libertà e non essere in nessun modo vincolato dall'infrastruttura. In aggiunta, è possibile scegliere liberamente di utilizzare vari tipi di macchine con configurazioni diverse da più punti di vista, come per esempio:

- Potenza computazionale (numero di core fisici/virtuali)
- Banda di rete
- Accelerazione grafica
- Storage locale del server

in modo da poter personalizzare al massimo il servizio.

1.2.2 PaaS

Nel modello PaaS il provider mette a disposizione del cliente una piattaforma pre-impostata che permette l'implementazione di applicazioni e servizi.

Una servizio di questo tipo è formato da macchine con sistemi operativi già installati e configurati, e che possono possedere altri strumenti utili allo sviluppo come DBMS, web server, IDEs, framework etc.

La differenza con IaaS è che qui l'utente non deve occuparsi dei dettagli di basso livello, come la virtualizzazione e la gestione del sistema operativo.

Questo ovviamente rende più facile e meno costoso il lavoro dell'utente, che non dovrà più avere esperienza nella gestione e configurazione di risorse hardware, ma, ovviamente, ne limita anche il potere decisionale privandolo di parte del controllo.

I servizi PaaS si possono dividere in:

- Programming Environment, come Django Framework
- Execution Environment, come Google App Engine e Microsoft Azure

La prima categoria definisce gli ambienti principalmente adibiti al supporto all'implementazione di codice, mentre la seconda quelli che si concentrano nell'eseguire del programmi astraendo dall'architettura sottostante.

1.2.3 SaaS

Se tutto ciò che il provider offre al cliente è l'utilizzo di un'applicazione, si rientra nella categoria SaaS. In quest'ultima, appunto, l'utente finale può utilizzare liberamente un software, mentre tutti i dettagli sottostanti sono gestiti dal provider.

Solitamente è necessaria la registrazione per poter utilizzare il servizio, ed essa può essere gratuita o a pagamento.

In questo paradigma il client ha bisogno solo delle skills necessarie ad utilizzare il programma e nient'altro, tuttavia ciò implica che egli non abbia nessun genere di controllo relativo all'architettura di basso livello.

Ad oggi i servizi SaaS sono moltissimi, dalle web-mail (Gmail), ai tool di organizzazione (Trello), ai fogli di calcolo (Google Docs), e hanno rivoluzionato il modo di utilizzare la rete.

1.3 Attori del cloud computing

Normalmente all'interno delle infrastrutture cloud si possono definire cinque tipi di attori:

- Cloud Provider
- Cloud Consumer
- Cloud Carrier
- Cloud Auditor
- Cloud Broker

che andremo a spiegare più in dettaglio qui di seguito.

Il Cloud Provider è colui che offre i servizi in grado di soddisfare le necessità del Consumer, per ogni modello di servizio (IaaS, PaaS, SaaS) egli offre le funzionalità di archiviazione dati e le interfacce che consentono la gestione ai clienti.

Il Consumer viene definito come un'entità che ha rapporti commerciali con il Provider e utilizza i suoi servizi. Come già accennato, il ruolo effettivo del Consumer cambia secondo il servizio utilizzato, infatti in IaaS ha una capacità di controllo molto elevata che diminuisce man mano spostandosi verso il SaaS.

Con Cloud Carrier si intende quel soggetto che fornisce al Provider i servizi di rete e trasporto, in modo da poter comunicare con il Consumer. Solitamente Carrier e Provider stipulano un SLA (Service Level Agreement) cioè un accordo

sulla qualità del servizio che il Carrier dovrà offrire. In base al SLA con il Carrier, il Provider potrà decidere che SLA utilizzare con i Consumer.

La figura del Cloud Broker entra in gioco quando i clienti non riescono a gestire in modo semplice il servizio cloud. Come indica il nome, il suo ruolo è quello di fare da intermediario tra Provider e Consumer alleggerendo i compiti del cliente. Per entrare più in dettaglio, un Broker può svolgere servizi di performance reporting e security enhancement, ma anche cercare di ottimizzare il costo della piattaforma combinando vari servizi, anche di Provider differenti.

Il Cloud Auditor è una terza parte indipendente che si occupa principalmente di monitoraggio. In particolare, svolge alcune funzioni di controllo tra cui, per esempio, la misurazione delle performance e del grado di privacy e sicurezza dell'implementazione. Verifica, inoltre, che il Provider rispetti gli standard.

1.4 Modelli di distribuzione

Le infrastrutture di cloud computing si possono dividere anche dal punto di vista della distribuzione, infatti possono esistere (figura 1.2[4]):

- Private Cloud
- Public Cloud
- Community Cloud
- Hybrid Cloud

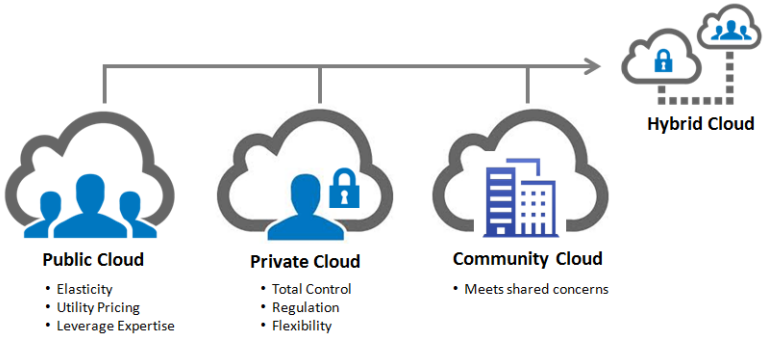

Figura 1.2: Modelli di distribuzione cloud

La differenza qui sta in chi può utilizzare i servizi e in chi li gestisce.

[4]da www.mytechlogy.com

I Private Cloud (o Internal Cloud) sono quelle strutture in cui il provider e gli utilizzatori fanno parte della stessa organizzazione o azienda. Lo scopo principale dei cloud privati è mantenere la ownership dei dati e gestire totalmente la sicurezza. Il compito di gestione della piattaforma può essere svolto dall'azienda stessa, da una terza parte o da una combinazione di essi.

Nel cloud pubblico, invece, l'infrastruttura è disponibile a chiunque ne voglia far uso, a fronte di un costo basato sull'utilizzo. I provider e gestori di tali servizi sono solitamente grandi aziende o enti pubblici.

I Community Cloud, in italiano Cloud Comunitari, sono piattaforme condivise da un insieme di organizzazioni con finalità comuni e che solitamente vengono gestiti da una o più delle aziende del gruppo, a volte con l'aiuto di una terza parte.

L'Hybrid Cloud è semplicemente una struttura con caratteristiche di due o più delle infrastrutture sopra descritte. Un possibile esempio può essere un sistema che a carico normale presenta le caratteristiche di un Private Cloud, ma, quando si presenta un picco temporaneo di richieste, scala utilizzando la potenza di un cloud pubblico, facendo però attenzione a non trasferire all'esterno dati critici per l'azienda. In ognuna di queste categorie, solitamente, le risorse vengono messe a disposizione attraverso un portale web cercando di rispettare al meglio il già citato principio dell' "On-Demand Self-Service"

1.5 Sicurezza nel cloud[5]

Le infrastrutture di cloud computing, fanno emergere alcuni nuovi rischi di sicurezza, la Cloud Security Alliance [2] li divide in:

- Abuse and nefarious use of cloud computing
- Malicious insiders
- Insecure Interfaces and API
- Shared technology issue
- Data loss or leakage
- Account service hijacking
- Unknown risk profile

[5]In parte basato su: "SQL + Cloud security" - Slide del corso di Sicurezza delle reti, Gabriele D'Angelo

La prima categoria si rifersce all'utilizzo illecito delle infrastrutture cloud. Essendo particolarmente semplice ottenere l'accesso a un servizio cloud, è possibile che degli attaccanti utilizzino i servizi a disposizione per condurre vari tipi di attacchi, tra cui, per esempio, produzione di spam e DoS. Le contromisure a questo rischio sono l'analisi del traffico di rete e l'irrigidimento delle procedure di registrazione.

Un altro rischio riguarda la presenza di individui con cattive intenzioni all'interno dei dipendenti del cloud provider. I cosiddetti malicious insider, infatti, sono persone che sfruttano le loro capacità di accesso ai dati per minacciare la sicurezza. Qui, le principali strategie di mitigazione del rischio, sono prestare più attenzione alle risorse umane sin dalle prime fasi di recruiting e stabilire dei protocolli per la notifica di situazioni pericolose.

Il rischio "Insecure Interfaces and API" si riferisce appunto alle API utilizzate dagli utenti per interagire con il servizio cloud. Queste interfacce sono fondamentali per l'utilizzo, e una loro potenziale falla di sicurezza potrebbe essere molto grave. Per evitare danni legati a questo problema si possono effettuare diversi tipi di manovre, come controllare periodicamente il modello di sicurezza delle API, migliorare le politiche di access control e utilizzare crittografia.

Molto spesso i servizi cloud, per scalare in maniera migliore, virtualizzano più risorse (CPU, GPU, cache etc.) che condividono la stessa macchina fisica, tuttavia capita che il livello di isolamento tra le risorse stesse non sia sufficiente per garantire la robustezza di un modello multi-tenant (definito più in dettaglio in 1.5.1). Per ovviare a questo problema ci sono diverse tattiche, dall'hardening del sistema operativo per aumentarne la sicurezza, al rafforzamento delle procedure di controllo degli accessi e autenticazione, al monitoring delle attività.

Le infrastrutture cloud vengono utilizzate anche per contenere dati sensibili, perciò deve essere minimizzato il rischio di perdita e condivisione non autorizzata degli stessi. Per fare ciò si utilizzano particolari policy di access control e si possono crittografare i dati sia durante la memorizzazione che durante il transito, in questo caso bisogna fare attenzione a implementare politiche robuste per la generazione, distribuzione e distruzione delle chiavi.

L'account hijacking è un tipo di attacco in cui l'attaccante riesce ad avere il pieno controllo di un account non proprio, ciò ovviamente mette a rischio tutte le risorse a cui quell'account ha accesso. Solitamente questo rischio si concretizza tramite il furto delle credenziali, le contromisure sono il divieto di condividere le stesse, l'autenticazione a più fattori e pratiche di intrusion detection.

L'ultimo rischio si riferisce al fatto che i clienti non hanno il totale controllo sui dettagli interni dell'infrastruttura cloud, i quali possono impattare sulla

sicurezza. Per combattere questo pericolo il provider può offrire dei log dettagliati o pubblicare parte dei dettagli implementativi, in aggiunta all'esecuzione di operazioni di auditing.

1.5.1 Modelli di tenancy

Le piattaforme cloud possono offrire ai clienti più modelli di tenancy, cioè di divisione delle risorse. I due principali sono:

- multi-instance
- multi-tenant

Nel primo modello il cliente ha il controllo esclusivo di una singola macchina/DBMS che quindi non viene condivisa con altri, in questo modo la sicurezza dipende dalle impostazioni del cliente sull'istanza. Nel modello multi-tenant invece, le macchine e/o i DBMS sono condivisi tra più utenti (o tenant), qui ogni utente utilizza un sottoinsieme delle capacità totali, cioè quelle a lui assegnate. Il mantenimento della sicurezza e dell'isolamento tra utenti è carico del provider.

Dal punto di vista pratico il modello multi-instance ha il vantaggio di poter controllare le politiche di sicurezza, mentre il multi-tenant permette di scalare in maniera molto più semplice.

1.6 Scelta di utilizzo del cloud

A questo punto ci si può porre la domanda:"Quando è conveniente utilizzare il cloud?", la risposta ovviamente non può essere univoca e dipende dagli obiettivi che gli utenti hanno. Per cui è sempre necessario effettuare un analisi costo-beneficio, in particolare per verificare se è più conveniente costruire un'infrastruttura in-house o affidarsi a un provider di servizi.

Per fare un esempio, assolutamente non esaustivo, dei parametri che possono influenzare la scelta, facciamo notare che le soluzioni in-house sono completamente customizzabili, flessibili e controllabili, tuttavia per utilizzarle è necessario un grande investimento iniziale e periodiche manutenzioni. I servizi cloud invece, rimuovono tutti i costi relativi allo startup, la manutenzione e l'energia consumata, a scapito di una maggiore rigidità e a un non particolarmente elevato grado di sicurezza.

Capitolo 2

Amazon Elastic GPUs

2.1 Premessa

Amazon Elastic GPUs (da qui in avanti EGPUs) è un servizio che permette di collegare dinamicamente una GPU virtuale a un'istanza EC2, per poter usufruire di potenza grafica senza dovere utilizzare istanze con GPU dedicata. Ciò garantisce un'esperienza più customizzabile per l'utente, insieme alla possibilità di ottimizzare i costi.

Figura 2.1: Logo EGPUs

Prima di entrare nel dettaglio degli scopi e della struttura di EGPUs è necessario introdurre alcuni concetti basilari, cosa che faremo di seguito.

2.1.1 Gestione di carichi di lavoro grafici

Al giorno d'oggi moltissime delle applicazioni utilizzate, sia a livello consumer che a livello business, hanno bisogno di accelerazione grafica dovendo gestire immagini (2D/3D) e video. Tra queste ci sono:

- Videogames
- Programmi di Computer aided design/engineering/modelling (CAD/-CAE/CAM)
- Software per architettura
- Software per imaging in ambito medico
- Applicazioni di simulazione
- Programmi per animazione e creazione di contenuti

Storicamente, per usufruire di queste applicazioni, si sono sempre utilizzate delle workstation, cioè computer o cluster con alta disponibilità di CPU e GPU in modo da permettere prestazioni ottimali. Questi sistemi sono solitamente molto potenti, ma anche molto costosi e difficili da mantenere.

Quasi sempre, i progetti che coinvolgono le categorie di software precedentemente citate, sono svolti in collaborazione tra più persone e utilizzando più di un programma. Prendiamo l'esempio del ciclo di sviluppo di un veicolo (figura 2.2)

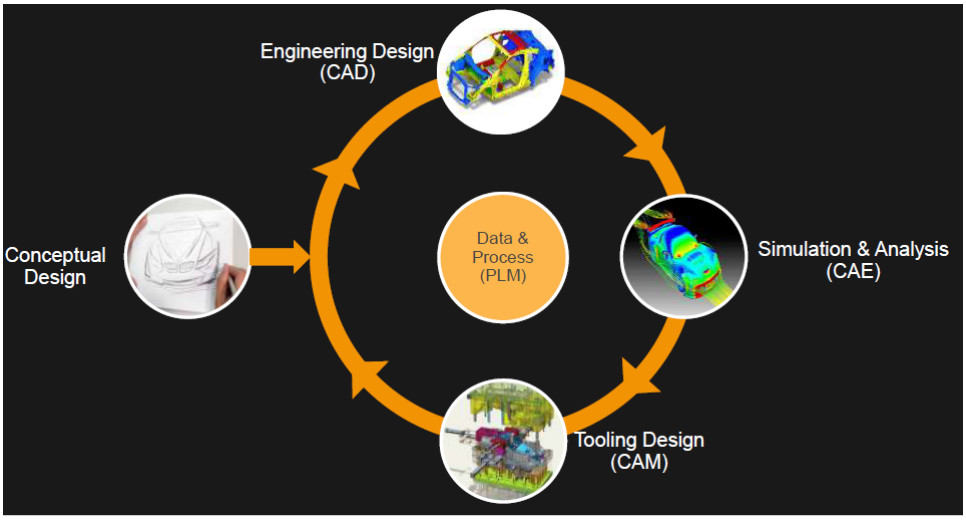

Figura 2.2: Utilizzo di software grafici durante lo sviluppo di un veicolo

Il passaggio dei dati tra le varie applicazioni e tra i vari utenti può non essere semplice, infatti molto spesso avviene tramite scambio fisico di file tra i designer, metodo che risulta molto scomodo. Alcuni strumenti, come i file-system distribuiti possono aiutare a condividere i file, ma il processo spesso

risulta comunque lento. Con le tecnologie di cloud computing è possibile centralizzare non solo i dati, ma anche la potenza computazionale e grafica, in questo modo, oltre a migliorare e velocizzare la cooperazione tra individui, non si ha più bisogno di costose workstation locali.

L'idea della remotizzazione delle GPU è quella di permettere che tutto il calcolo grafico necessario a un'applicazione possa essere svolto dall'infrastruttura cloud, in modo che il cliente riceva solamente il risultato finale e possa interagire con esso. La figura 2.3 mostra chiaramente questo processo

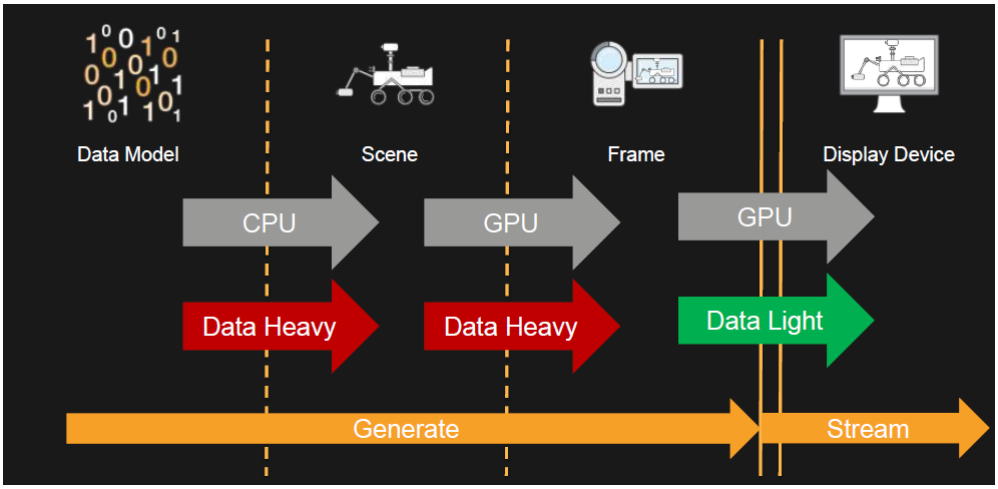

Figura 2.3: Flusso dati in un'applicazione con GPU remotizzata

Dai dati grezzi del modello si crea la scena generale utilizzando la CPU, in questo passaggio i dati sono pesanti perché contengono tutti i dettagli del progetto. Dalla scena si renderizzano i vari frame, per esempio quelli che permettono di visualizzare un video o di muovere un modello 3D, anche in questo caso la quantità di byte da processare con la GPU è molto alta. L'ultimo passaggio riguarda l'invio dei frame generati, che sono fondamentalmente immagini, al dispositivo che permette di visualizzarli e che, solitamente, è il pc del cliente, questo stream di dati risulta quello più leggero.

2.1.2 APIs grafiche

Un API grafica [23] definisce il modo in cui le applicazioni devono interagire con l'hardware per utilizzare le funzioni che consentono il disegno e il rendering di immagini e video. Quasi sempre questo genere di interfacce sono sviluppate come un driver software capace di interagire con la GPU, sebbene sia possibile utilizzarle anche in CPU, per esempio nei dispositivi embedded. Nel seguito

parleremo delle API più utilizzate al giorno d'oggi cioè OpenGL [20][22][28], DirectX [24] e OpenCL [7][25][26].

Ci concentreremo di più su OpenGL, essendo quest'ultima l'unica supportata da EGPUs.

OpenGL

OpenGL è una specifica, gestita dal gruppo Khronos, utilizzata per sviluppare applicazioni 2D e 3D, permette di comunicare con la GPU per ottenere accelerazione hardware.

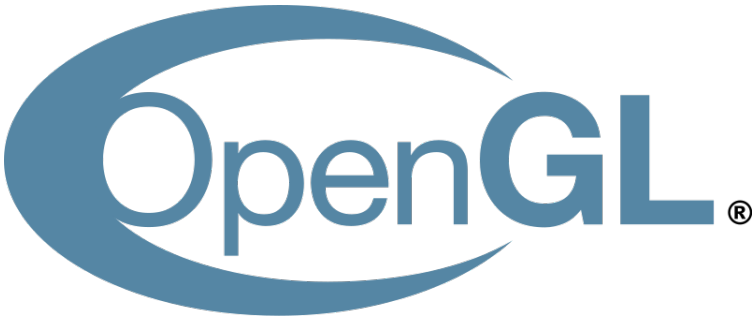

Figura 2.4: Logo OpenGL

Definisce ad alto livello come dovrebbero essere gestite la grafica e il gpu computing, e quindi quali funzionalità andrebbero supportate, tuttavia non fornisce codice concreto, infatti ogni vendor può implementare a sua discrezione la specifica. Non avendo implementazione è quindi da considerarsi indipendente sia dalla piattaforma che dai sistemi operativi. Proprio perché ogni vendor implementa OpenGL in modo diverso, il codice non può essere considerato strettamente open-source, tuttavia, esistono delle librerie, come Mesa3D, che sono disponibili pubblicamente.

OpenGL supporta tutte e sole le primitive geometriche offerte dalle GPU odierne (punti, linee e triangoli) e non strutture più complesse come case o automobili. Perciò gli sviluppatori dovranno costruire i loro modelli partendo da queste semplici funzioni. Bisogna specificare che sono disponibili delle librerie basate su OpenGL che mettono a disposizione le funzioni per creare modelli più complessi basandosi sulle primitive, in questo modo il lavoro dello sviluppatore è semplificato. Per mantenere l'indipendenza dai sistemi operativi, inoltre, OpenGL non possiede nessun costrutto che gli permetta di prendere input esterni o di gestire le finestre, questa potrebbe essere vista come una grande limitazione, ma è possibile, in modo semplice, collegare altre librerie a OpenGL in modo da sopperire a queste mancanze. La specifica gestisce anche le interazioni con la rete, infatti è possibile separare un'applicazione grafica

in parte client e parte server, per esempio quest'ultimo potrebbe svolgere una complessa simulazione mentre il client limitarsi a mostrare i risultati.

OpenGL al giorno d'oggi è considerata l'API di grafica più semplice in circolazione [20], ed essendo cross-platform è anche largamente utilizzata, per il futuro, il gruppo Khronos sta sviluppando Vulkan, che dovrebbe essere di fatto il successore di questa specifica.

La prima versione di OpenGL è stata rilasciata il 30 Giugno 1992, da quel momento in poi ogni versione successiva ha esteso la specifica iniziale, incorporando man mano le estensioni che si diffondevano tra i vari vendor. Normalmente, dopo la prima versione, ogni nuova release N è stata composta dalle caratteristiche della N-1 con alcune nuove estensioni, per esempio la versione 4.3 contiene le funzionalità della 4.2 con delle particolari aggiunte.

DirectX

DirectX è una collezione di APIs sviluppate da Microsoft per i sistemi Windows, è al momento uno dei principali rivali di OpenGL. La X del nome è un placeholder generico per indicare i vari pacchetti che la compongono, cioè:

- DirectDraw, per la grafica 2D
- Direct3D, per la grafica 3D
- DirectSound, per il suono 2D
- DirectSound3D, per il suono 3D
- DirectMusic, per la gestione della musica
- DirectPlay, per la gestione del multiplayer
- DirectInput, per i device di input

Il principale ruolo di DirectX è fare in modo che alcuni programmi, tra cui specialmente i giochi, possano accedere direttamente all'hardware della macchina in modo da accelerare workload grafici e di compute tramite GPU e con supporto di shader [31][32] e GPGPU compute. Inoltre, in caso un dispositivo hardware non sia disponibile, DirectX è in grado di simulare le sue funzioni lato software, in modo che si possano evitare eventuali errori.

La principale differenza tra OpenGL e DirectX è che quest'ultimo non è cross-platform, ciò può essere visto sia come vantaggio che come svantaggio, infatti, non dovendo gestire le differenze tra un sistema operativo e l'altro, è possibile ottimizzare il codice in modo migliore. Inoltre, mentre OpenGL è un API prettamente grafica, DirectX controlla molti più aspetti relativi all'hardware.

OpenCL

OpenCL è un tipo di API diversa da quelle precedentemente citate, infatti, pur venendo utilizzata come tramite tra l'hardware e le applicazioni, ha uno scopo particolare, cioè il GPGPU (General purpose computing on graphical processing units). Questo tipo di computazione viene utilizzato dai programmi che sfruttano i molteplici processori delle GPU per svolgere calcoli paralleli non solo grafici, un esempio si può trovare nelle applicazioni che avviano compute workloads mentre procedono al rendering di un immagine.

Il principale vantaggio di questa API sta nel fatto di semplificare la gestione dei calcoli paralleli, permettendo di astrarre da dettagli implementativi, ciò avviene perché OpenCL, similmente a OpenGL, è una specifica di alto livello che mira ad essere completamente platform/vendor/hardware independent.

Il modello quindi, permette agli sviluppatori di sfruttare con facilità tutta la capacità computazionale che i loro dispositivi possano fornire, siano essi workstation, cluster o dispositivi mobili.

Anche OpenCL è mantenuta dal gruppo Khronos, e pur non essendo strettamente un API grafica è compatibile con molte di esse, tra cui OpenGL.

2.2 Scopo di Elastic GPUs

Come già accennato, lo scopo di EGPUs è permettere di collegare a delle istanze EC2 un'accelerazione grafica, in modo da poter eseguire applicazioni che richiedano il supporto di una GPU hardware. Si fa notare che si può scegliere liberamente il tipo di istanza EC2 da utilizzare, in modo da poter configurare anche le quantità di CPU e RAM più adatte, superando le limitazioni delle istanze grafiche autonome.

Al momento le GPU elastiche supportano OpenGL fino alla versione 4.3 e possono raggiungere gli 8GB di memoria grafica, ciò le rende adatte ad applicazioni che generalmente sono concepite per consumer e workstation GPU, come per esempio, la virtualizzazione HPC, i giochi, la progettazione industriale e i virtual desktop. Le EGPUs sono la migliore scelta anche quando ci sono dei software che utilizzano grandi quantità di CPU e RAM, ma che possono beneficiare di una quantità di GPU.

2.2.1 Altre istanze grafiche in EC2

Prima di EGPUs gli utenti che intendevano utilizzare accelerazioni grafiche in EC2 dovevano utilizzare istanze G2 o G3. Entrambi i tipi di istanze posseggono delle schede grafiche di alta gamma come le NVIDIA GRID K520 o le NVIDIA TESLA M60, questo permette loro di gestire con facilità qualsiasi applicazione grafica, comprese quelle citate in 2.1.1. Tuttavia hanno quantità di CPU e RAM fissate e non troppo alte, per cui funzionano in maniera migliore in applicazioni che utilizzano molto la GPU e poco il processore. Entrambe le istanze supportano DirectX, OpenGL, CUDA e OpenCL. Al giorno d'oggi le G3, che stanno rimpiazzando le G2, vengono usate per batch rendering, realtà virtuale e generazioni di effetti video.

Dimensioni GPU elastica	Memoria GPU
eg1.medium	1 GiB
eg1.large	2 GiB
eg1.xlarge	4 GiB
eg1.2xlarge	8 GiB

Nome	GPU	vCPU	Memoria (GiB)	Memoria GPU (GiB)
g3.4xlarge	1	16	122	8
g3.8xlarge	2	32	244	16
g3.16xlarge	4	64	488	32

Figura 2.5: Dettagli di EGPUs e G3

2.2.2 Vantaggi

Le EGPUs sono nate in quanto permettono di avere una lunga serie di vantaggi rispetto alle altre istanze grafiche standalone. In primis, le G2 e le G3 hanno quantità fisse di CPU e RAM per cui si possono avere due situazioni problematiche:

- Dover sovraprovisionare la GPU per avere il giusto ammontare di CPU o RAM
- Dover sovraprovisionare la CPU e la RAM per avere la giusta quantità di GPU

questo overprovisioning porta a uno spreco di risorse (per le quali si paga). Con Elastic GPUs, invece, si può scegliere liberamente l'istanza da utilizzare tra quasi tutte quelle fornite da EC2 e poi collegare una qualsiasi EGPU della dimensione che si preferisce. In questo modo si utilizza solo ciò che è necessario all'applicazione. Un altro vantaggio importante delle GPU elastiche è il costo, infatti, il loro prezzo, unito a quello delle macchine a cui sono collegate, risulta spesso molto più basso di quello delle istanze con grafica standalone. Come è possibile vedere dalla figura 2.6, l'utilizzo di un'istanza di classe standard con una EGPU può portare un grosso beneficio economico, che può raggiungere l'85% di spesa in meno.

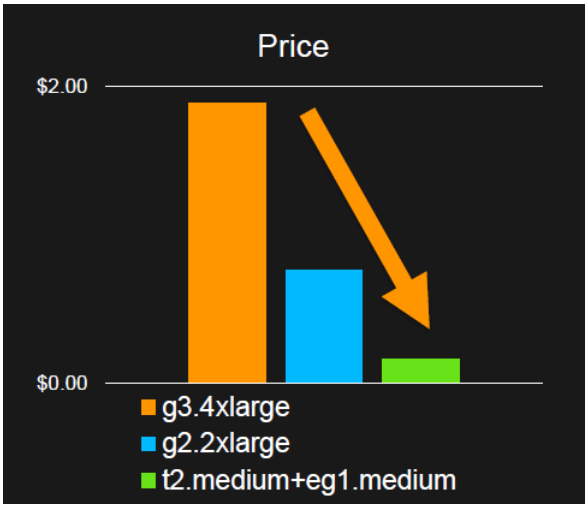

Figura 2.6: Comparazione tra i prezzi delle possibili istanze con accelerazione grafica in EC2

Dal punto di vista delle performance, inoltre, si nota come la relativamente limitata potenza delle EGPUs non impatti in maniera significativa l'utilizzo di molti tipi di applicazioni, infatti dai 20 FPS[1] in su, l'occhio umano non riesce a percepire quasi nessuna differenza, perciò l'utilizzo di istanze più performanti sarebbe inutile.

L'utilizzo di EGPUs è possibile attraverso vari tipi di protocolli di remotizzazione, tra cui DCV, RDP e VNC.

Riassumendo, le EGPUs hanno il vantaggio di poter essere collegate a quasi tutte le istanze EC2, permettendo di ottimizzare i costi evitando overprovisioning, ma sempre mantenendo prestazioni al livello di una workstation.

2.2.3 Limitazioni

Le GPU elastiche hanno attualmente alcune limitazioni, innanzitutto l'utente che le utilizza non ha il totale controllo degli aspetti relativi alla scheda grafica, come per esempio accade nelle G3, e quindi non può personalizzare al massimo la stessa, inoltre, nelle EGPUs non si può avere accesso all'hardware encoding.

A livello di framework le istanze grafiche standalone supportano molteplici APIs, spesso con le ultime versioni, mentre le EGPUs utilizzano solo OpenGL (fino alla versione 4.3).

Come già detto, a livello di performance, in applicazioni generiche EGPUs si comporta molto bene, ma in caso di calcoli GPU-intensive le G2/G3 sono ancora la scelta migliore. Inoltre, come spiegheremo in dettaglio più avanti, le istanze con EGPUs accedono alla memoria grafica attraverso la rete, questo introduce una latenza non nulla, che può impattare la user-experience.

[1] fotogrammi al secondo

2.3 Struttura

Per parlare della struttura e del funzionamento di EGPUs ci baseremo sulla seguente figura 2.7.

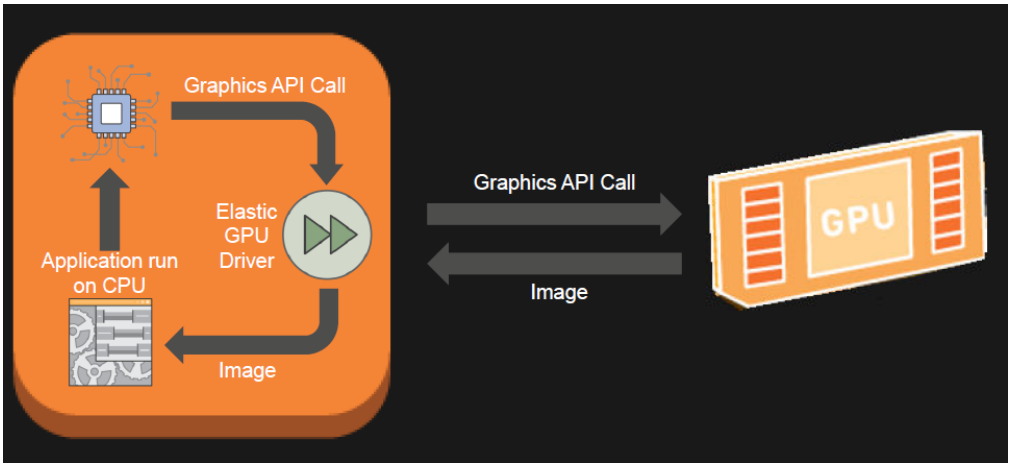

Figura 2.7: Struttura Elastic GPUs

Come già accennato, la GPU è fisicamente separata dalla macchina che viene utilizzata per far girare l'applicazione, quindi possiamo definire due componenti:

- La User Machine (o UM), cioè la macchina EC2 che l'utente ha selezionato.
- La Graphics Machine (o GM), cioè la struttura che contiene la scheda grafica che viene utilizzata.

UM e GM sono collegate da un rapporto 1-1. Essendo la GM non collegata fisicamente alla macchina, non è possibile vedere la scheda grafica a livello di device del sistema operativo, ma è possibile controllare vari parametri dalla console EC2 e con altri particolari metodi.

L'UM contiene il componente fondamentale di EGPUs, cioè il suo driver, che controlla la comunicazione con la GM, esso, al contrario della scheda grafica, è mostrato all'utente come graphic renderer.

Quando un'applicazione fa una chiamata OpenGL sull'UM, quest'ultima viene intercettata dal driver, che poi invia il comando alla GM attraverso la rete, utilizzando un apposito canale chiamato ENI (Elastic Network Interface). Dopo aver ricevuto le istruzioni, la GM le computa e genera l'immagine risultante, successivamente inviata indietro all'UM sempre attraverso la rete. A questo punto sull'istanza viene visualizzata a schermo la figura.

Questo procedimento deve essere svolto per ogni comando richiesto dall'applicazione e quindi moltissime volte al secondo, per cui è molto importante minimizzare la latenza tra UM e GM, ciò viene fatto mantenendo quest'ultima nella stessa AZ[2] dell'istanza. Ogni istanza EC2 ha dei limiti di banda, ciò significa che anche il tipo di UM può influire sulla velocità di comunicazione con la GM, per cui è possibile che un'istanza EC2 più potente permetta un utilizzo più fluido di EGPUs. Sempre dal punto di vista delle performance, la GM adotta un meccanismo particolare, se le richieste che gli arrivano sono semplici e si riescono a generare molti FPS, la GM ne invia indietro solo 24, ciò avviene per non intasare la rete e perché, come già accennato, non porta a cambiamenti sensibili nella user experience.

Il driver di EGPUs ha il totale controllo delle chiamate OpenGL e può arbitrariamente decidere se inoltrare o no una chiamata o se modificarla, questo può avvenire in caso di chiamate troppo pesanti o non supportate. Ciò riprende lo scopo di EGPUs, cioè fornire agli utenti che utilizzano applicazioni grafiche (non troppo GPU-intensive) la miglior esperienza possibile.

Le immagini generate e le computazioni richieste, dunque, viaggiano attraverso la rete, ciò potrebbe risultare un rischio di sicurezza se non si presta attenzione a come viene effettuata la comunicazione. In EGPUs tutto il traffico con la scheda video resta sempre all'interno della VPC[3] dell'utente e quindi è visibile solo a quest'ultimo.

[2]Un AZ o Availability Zone, è una particolare porzione della rete EC2 che contiene macchine geograficamente vicine e collegate con infrastrutture a bassa latenza

[3]Virtual Private Cloud, sezione di cloud privata, logicamente separata dalle altre e personalizzabile

2.3.1 Utilizzare Elastic GPUs

Utilizzare EGPUs è molto semplice, infatti, il progetto è completamente integrato con la console EC2, con l'AWS SDK e l'AWS CLI, in modo che gli utenti possano usarlo come preferiscono. Una limitazione importante sta nel fatto che è possibile usufruire di EGPUs solo sotto macchine Windows.

Per lanciare un istanza con EGPUs basta quindi scegliere il tipo dell'istanza e la size della GPU che si desidera avere. Per controllare il funzionamento del driver e della scheda grafica si possono utilizzare sia le metriche pubblicate nella console EC2 e in Cloudwatch, sia i sanity check mostrati nel manager di EGPUs (figura 2.8), presente nella barra delle attività.

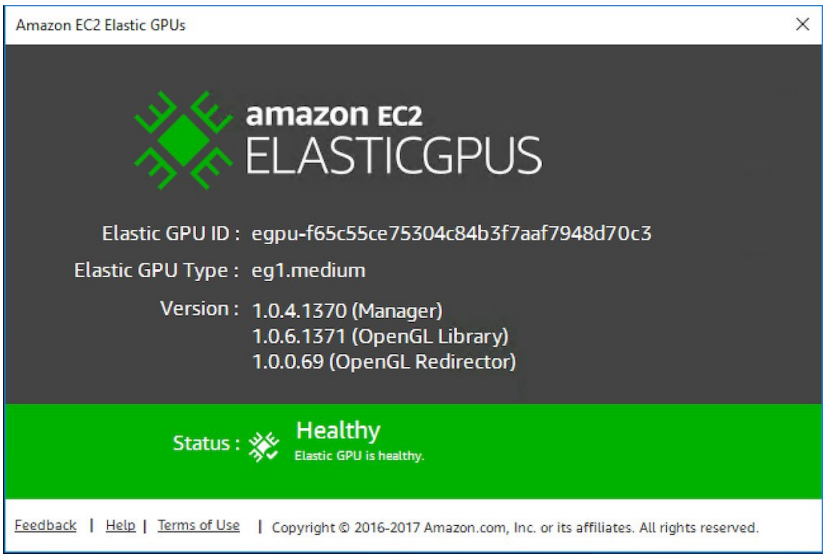

Figura 2.8: Elastic GPUs Manager

Capitolo 3

Metodologie di testing

Di seguito si darà una visione generale delle metodologie di testing utilizzate al giorno d'oggi.

3.1 L'importanza del testing

Ogni programmatore, anche il più bravo, commette sempre degli errori, essi possono essere causati da vari motivi, tra i quali: disattenzioni, poca chiarezza nelle specifiche e scarsa conoscenza dell'ambiente o del linguaggio. Ovviamente è necessaria una maniera per accorgersi di questi errori, in modo da correggerli al più presto ed evitare perdite di tempo. Questo è ancora più importante in progetti di alto livello, dove anche un piccolo errore può portare ad un grande disservizio nella rete, impattando molteplici clienti.

Il testing, quindi, è fondamentale in ogni progetto, dal più piccolo al più grande, e può arrivare ad occupare una gran quantità di tempo. Molto spesso, tuttavia, questo processo è sottovalutato dai programmatori, che non vogliono perdere tempo a scrivere codice "non operativo". Questo accade soprattutto agli sviluppatori alle prime armi e a coloro che, durante il loro percorso di studi, non hanno avuto l'occasione di approfondire l'importanza di questo argomento.

Da dieci anni a questa parte il tema del testing ha avuto molta più visibilità in letteratura e sta venendo pian piano assimilato nei piani di studio delle università. In aggiunta, gli strumenti di testing a disposizione dei programmatori sono molto più potenti e facili da utilizzare, agevolando quindi la diffusione delle best practices riguardo alla verifica del codice.

Prima di procedere ulteriormente si da una definizione di "Software Testing" citando [4]. "Il software testing è un processo o una serie di processi creati in modo tale da verificare che il codice svolga il compito per cui è stato sviluppato e non svolga niente di non desiderato. Il software deve essere predicibile e consistente, e non deve offrire sorprese agli utenti"

3.1.1 Testing nei vari modelli di sviluppo

Nei vari modelli e metodologie di sviluppo, il testing non ha sempre avuto lo stesso ruolo. Agli inizi dell'ingegneria del software, con l'approccio a cascata, i test venivano scritti e svolti solo dopo aver completato la fase di implementazione, ciò poteva provocare parecchi problemi, tra cui la difficoltà di correggere gli errori rapidamente, soprattutto nel caso in cui gli stessi mostrassero una mancanza nel design. L'approccio era quindi troppo rigido per essere cost-effective.

Ai giorni d'oggi, con l'Agile, il testing è presente in praticamente tutte le fasi del processo di sviluppo, in modo da cercare di assicurare la più alta correttezza del codice. L'importante in questo caso è l'automazione, in modo che gli sviluppatori non perdano tempo a effettuare lunghi test manuali e non deterministici, potendosi concentrare su altri compiti.

3.2 Test-Driven Development

Il Test-Driven Development (TDD di seguito), come dice il nome, è lo sviluppo di codice guidato dai test, questo modello è basato su delle semplici regole che andiamo a dettagliare di seguito (da [3]):

- Non scrivere codice in produzione prima di aver uno unit test che fallisce;
- Non scrivere più codice di test di quello necessario ad ottenere un fallimento, considerando l'impossibilità di compilare un fallimento;
- Non scrivere più codice di produzione di quello necessario a far passare i test case che falliscono.

In questo modo il codice di production e quello di testing vanno di pari passo aumentando la coverage[1].

Il processo di scrittura di codice diviene quindi il cosiddetto "Red-Green-Refactor cycle", in quest'ultimo, partendo da un test che fallisce (Red), si scrive il codice che lo fa passare (Green) per poi sistemare quest'ultimo (Refactor); il tutto normalmente dovrebbe avvenire in 10-20 minuti.

[1]Percentuale di codice attraversato dai test rispetto al totale del codice, più dettagli in 3.2.1

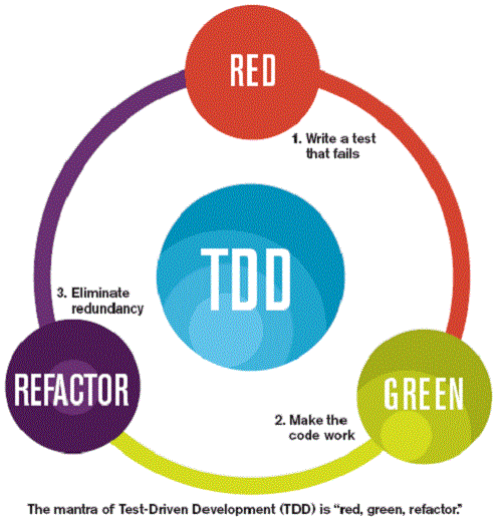

Figura 3.1: Test Driven Development

Il TDD, quindi, incoraggia il refactoring e la modifica del codice, infatti, avendo delle suite di test pronte, gli sviluppatori non avranno paura ad effettuare i cambiamenti necessari ad aumentare flessibilità, manutenibilità e riusabilità del codice, potendosi immediatamente accorgere di eventuali errori.

L'utilizzo massivo dei test significa anche che il codice degli stessi possa diventare molto voluminoso, e quindi difficile da mantenere, per evitare problemi è necessario tenerlo più pulito possibile, considerandolo allo stesso livello del production code.

Avere dei test scarsamente mantenibili può essere considerato peggiore di non avere nessun test, infatti, gli stessi dovranno cambiare continuativamente lungo la vita del progetto, e uno sforzo troppo alto nella modifica potrebbe portare gli sviluppatori a:

- Non eseguire i test;
- Non considerare i fallimenti dei test;
- Perdere moltissimo tempo per sistemare i test.

Rendendo di fatto la scrittura dei test stessi inutile.

Per mantenere le suite di testing pulite la cosa più importante è la leggibilità, un buon test deve essere leggibile facilmente e in modo rapido, senza confondere lo sviluppatore sullo scopo dello stesso. Non è sempre semplice promuovere la readability; un approccio spesso usato è costruire delle API di alto livello che vengano utilizzate dai test, in modo da nascondere molta della complessità sottostante, non necessaria alla comprensione. In questo modo si

vengono a creare dei Domain Specific Languages (DSLs) utilizzati soltanto per la scrittura dei test.

Alcuni linguaggi moderni, come Scala e Kotlin, hanno caratteristiche che permettono di aumentare al massimo la leggibilità delle suite, tra le quali la possibilità di utilizzare linguaggio naturale nel nome del test (e.g. "create leader and member, add task and change task status") o il supporto alla scrittura di DSLs.

Oltre alla leggibilità i test dovrebbero avere le cosiddette caratteristiche FIRST:

- Fast
- Indipendent
- Repeatable
- Self-Validating
- Timely

I test devono quindi essere, veloci, indipendenti tra loro, ripetibili, autovalidanti e scrivibili velocemente.

La velocità di un test è fondamentale, infatti, se un test è lento, gli sviluppatori saranno meno portati a lanciarlo e quindi sarà più difficile scovare eventuali errori.

I test devono essere indipendenti l'uno dall'altro, in modo da poter essere lanciati separatamente e in qualsiasi ordine. Se si hanno troppe dipendenze è possibile che si generino fallimenti a catena, che rendono difficile l'individuazione dei problemi.

Il determinismo dei test in ogni ambiente è fondamentale, in mancanza di questa caratteristica potranno generarsi falsi positivi e falsi negativi, che pian piano spingeranno i programmatori a non eseguire i test.

Un test ovviamente deve dire in automatico se ha ottenuto il risultato sperato o no, nessuna azione (e.g. lettura di log) deve essere necessaria da parte del programmatore per ottenere il risultato.

Anche la velocità di scrittura delle prove è molto importante, infatti, se un test richiede troppo tempo per essere scritto, è molto probabile che non venga scritto o che venga implementato male, abbassando la coverage dell'intero progetto.

I principi del TDD indicati in questa sezione si riferiscono normalmente agli Unit Test, cioè i test sviluppati per una singola funzionalità, tuttavia molti varranno anche per le categorie che definiremo in 3.3.2.

3.2.1 Code Coverage

La code coverage viene solitamente definita come la percentuale di codice attraversato dei test rispetto al totale della code base, tuttavia non esiste una definizione precisa, infatti bisogna specificare esattamente cosa significa "coprire" una parte di codice e a che livello (modulo, classe, metodo etc.).

La coverage normalmente viene utilizzata per identificare parti non testate dell'applicazione, tuttavia, citando Dijkstra ricordiamo che "Il testing mostra la presenza di errori, non la loro assenza", per cui, anche un codice con 100% di coverage, non sarà immune da bugs. Lo scopo non è quindi aumentare al massimo la coverage, ma testare le parti appropriate dell'applicazione.

3.2.2 Test doubles

I test doubles sono oggetti di test che sostituiscono le implementazioni reali, nel TDD sono fondamentali per riuscire a sviluppare test velocemente e continuativamente. Per esempio, per testare una classe A che contiene un oggetto della classe B ci sono due modalità:

- Creare il test per la classe A e successivamente creare la classe A e la classe B;
- Creare il test per la classe A e successivamente creare la classe A utilizzando un test double per la classe B.

Il secondo approccio è migliore soprattutto quando l'implementazione della classe B risulterebbe particolarmente lunga. Un altro vantaggio sta nella possibilità di non scrivere il codice relativo a B prima che venga scritto il test di B, rimanendo quindi fedeli ai principi del TDD.

Esistono vari tipi di test doubles:

- Dummy, oggetti passati, ma che non vengono mai utilizzati, solitamente servono a riempire liste di parametri;
- Stubs, implementazioni semplici, con metodi che non svolgono alcuna funzione e ritornano valori di default;
- Fake Objects, oggetti funzionanti, ma che utilizzano varianti semplici (algoritmi meno efficienti, DB in RAM);
- Test Spies, oggetti "spia" simili agli stub, ma che loggano qualsiasi chiamata venga fatta loro;

- Mocks, come i test spies, ma che hanno comportamenti particolari (e diversi dall'implementazione reale) sotto certe circostanze (e.g. "se la data è x usa l'algoritmo y e inserisci z nel log").

I test doubles hanno altre funzioni oltre a quella di velocizzare il TDD, infatti servono anche per rendere deterministici dei comportamenti randomici, per isolare il codice dalle proprie dipendenze, per osservare proprietà altrimenti invisibili e per simulare particolari condizioni difficili da ottenere normalmente.
Il tutto per riuscire ad individuare in modo migliore gli errori.

3.3 Classificazione del testing

Il software testing può essere classificato in più maniere, di seguito andremo a definirne vari tipi, con punti di vista e granularità differenti.

3.3.1 Prospettive di testing

Le prospettive di testing sono equiparabili a delle scuole di pensiero, e definiscono ad alto livello cosa viene testato e come. Riprendendo [11] le dividiamo in cinque categorie:

- Analytic school;
- Factory school;
- Quality school;
- Context-Driven school;
- Agile school.

La scuola analitica considera i programmi come artefatti logici soggetti alle leggi matematiche, per cui il testing diventa un compito rigoroso, oggettivo e formale. Per poterlo applicare è necessario avere requisiti molto precisi, in modo che coloro che svolgono i test possano assicurarsi che il software vi aderisca, per esempio il requisito "Il tempo di risposta deve essere accettabile da un utente" non è sufficiente per testare in modo analitico, mentre "Il 95% delle risposte deve arrivare entro 30 millisecondi" è decisamente più adatto. La code coverage nasce da questa scuola, in quanto capace di misurare in modo oggettivo la quantità di codice testata.

Con Factory school si intende quell'approccio dove lo sviluppo del software è considerato un progetto e i test un modo per misurare l'andamento dello

stesso. Il testing diventa quindi un'attività quasi manageriale e va pianificato, gestito e programmato oltre a verificare che sia cost-effective. Questa scuola è solitamente utilizzata in grossi progetti IT e incoraggia l'utilizzo di best-practices e l'acquisizione di certificazioni.

Il terzo punto di vista si basa sulla qualità, in questo caso il testing diventa la maniera per assicurarsi che gli sviluppatori stiano seguendo un buon processo di qualità e che non scrivano codice sporco. Solitamente esiste la figura del GateKeeper, cioè colui che controlla il codice e decide se il software sta rispettando i criteri di qualità richiesti. Questo modello viene utilizzato in organizzazioni sotto stress, poiché considerato meno alienante degli altri.

La Context-Driven school vede al suo cardine gli stakeholder, cioè le persone interessate al progetto. Un bug è definito tale solo se può risultare un errore per uno stakeholder. Si definisce quindi il contesto del software e si vanno a testare le funzionalità considerando questo contesto, lo stesso potrà poi cambiare durante il progetto, portando il testing ad essere un processo flessibile e adattabile.

L'ultima scuola è quella utilizzata nelle metodologie Agile. Il testing diventa un modo per aiutare il cambiamento e per capire quando una "user-story"[2] è finita. Questo approccio ha dato alla luce il già citato TDD e si concentra soprattutto sull'utilizzo di framework per automazione del testing. Sta diventando uno degli approcci prevalenti delle aziende IT.

3.3.2 Tipi di testing

Le prospettive di testing danno solo un punto di vista ad alto livello della materia, di seguito entriamo in maggiore dettaglio definendo molte delle possibili sfaccettature dell'ambito.

I test software possono essere technology-oriented o business-oriented, nel primo caso vengono definiti "prove di verifica" e difatti verificano che il software funzioni dal punto di vista tecnologico in tutti i possibili casi. La seconda categoria definisce le "prove di validazione" che appunto validano il completamento dei requisiti. Si nota come il testing tecnology-oriented si più adatto alla Analytic school, mentre il business-oriented alla Context-Driven e alla Factory.

Il testing può essere statico, se si effettua a livello del codice, o dinamico, se si effettua a run-time, al giorno d'oggi quello dinamico è molto più diffuso anche se lo statico non è del tutto scomparso.

Ovviamente un ulteriore divisione sta tra testing manuale e automatico, il primo viene eseguito effettivamente da un essere umano e può essere una simulazione d'esecuzione (test monkey) o un ispezione del codice. Il testing

[2]Descrizione ad alto livello di una funzionalità utile al raggiungimento di un obiettivo di business

automatico, invece, viene svolto da framework o script e quindi senza l'intervento umano, è di gran lunga il più utilizzato e viene considerato il miglior tipo tra i due, poiché introduce meno errori. Inoltre è uno dei cardini dell'Agile school.

Livelli di testing

Definiti questi dettagli bisogna specificare la granularità (scope) del testing, solitamente questi livelli si dividono in:

- Unit
- Integration
- System
- Acceptance

Lo unit testing comprende le prove che vengono svolte a livello di unità/funzionalità, è il testing di più basso livello e il cardine del TDD. Uno degli strumenti più utilizzati, in ambito Java, è JUnit[3] (figura 3.2[4]).

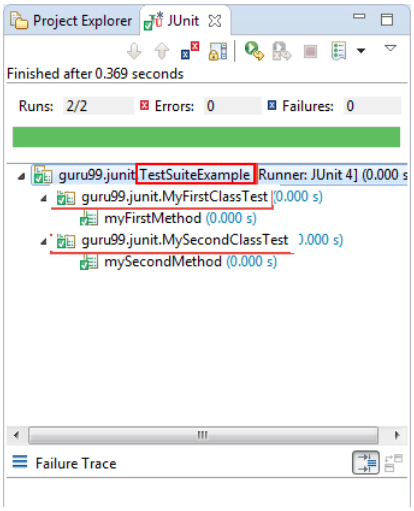

Figura 3.2: Test JUnit

[3]https://junit.org/junit5/
[4]da: https://www.guru99.com

L'integration test si ha quando si cercano di testare più funzionalità tra loro interconnesse, per verificare che il comportamento congiunto sia corretto, è quindi il livello superiore allo unit testing. JUnit è utilizzabile anche a questo livello, ma esistono molti altri strumenti, da scegliere in base alle proprie necessità.

Sia il system che l'acceptance testing vengono svolti sull'intero sistema sviluppato, ma in due maniere differenti. Il primo metodo si concentra sulla correttezza tecnologica del software e quindi cerca di verificare che "il prodotto sia costruito nel modo giusto", perciò si controlla che tutte le parti funzionino come dovrebbero e che rispettino i requisiti non funzionali (sicurezza, performance, etc.). L'acceptance testing, invece, è in stretta relazione con la Context-Driven school e si basa sui criteri di accettabilità definiti con gli stakeholders, quindi si concentra di più nel "costruire il prodotto giusto" per i clienti. Framework come Cucumber[5] o FitNesse[6] sono adatti a questo tipo prove.

Regression testing

Il regression testing [6], o testing di regressione, è un particolare tipo di test che normalmente si svolge dopo la messa in produzione del software, il suo scopo è verificare che una modifica al codice non abbia avuto effetti dannosi sul resto del programma. Solitamente il processo di regression testing inizia con la scoperta di un errore nell'applicazione sviluppata, dopo di che avvengono i seguenti passi:

- Individuazione dell'errore
- Correzione dell'errore
- Run delle test suite a disposizione

A questo punto, se i test hanno dato esito positivo, si procede a creare un test che simuli la situazione di errore appena riscontrata e risolta, in modo da poterlo aggiungere alle suite per il futuro. Successivamente il fix viene rilasciato in produzione. Se, invece, si è generato un fallimento, è presente una regressione, ciò significa che il fix prodotto per correggere l'errore ha provocato danni nel software, facendolo quindi "regredire". In questo caso è necessario rivedere il codice prodotto, modificandolo in modo da far passare le test suite presenti. Si fa notare che è importante scegliere se eseguire tutte le test suite a disposizione o no, infatti questo potrebbe portare via molto tempo e non essere

[5] https://cucumber.io/
[6] http://docs.fitnesse.org/FrontPage

sempre necessario, soprattutto quando la correzione è strettamente relativa a un solo modulo dell'applicazione.

I regression test vengono quindi svolti durante la manutenzione del software e si differenziano in due tipi in base al genere della stessa. I progressive regression test vengono svolti durante le fasi di manutenzione adattiva o perfettiva e in generale quando è presente una modifica alle specifiche del software. I corrective regression test, invece, sono relativi alla manutenzione correttiva e quindi a quelle modifiche al codice che non ne impattano la struttura interna. Dato che la manutenzione adattativa e perfettiva normalmente avvengono a cadenze regolari, anche i progressive regression test seguono questo andamento, mentre le correzioni devono essere svolte appena si incontra un errore, e quindi non ci sono intervalli fissi per questo tipo di test.

Whitebox vs Blackbox

Durante il testing bisogna scegliere se considerare i componenti del software come delle scatole chiuse di cui non si conoscono i dettagli o, al contrario, concentrarsi anche sull'implementazione interna. Questa scelta, che normalmente viene effettuata a livello di integration o system testing, descrive rispettivamente le categorie di blackbox e whitebox testing. Il blackbox testing viene definito anche testing funzionale, in quanto verifica che tutte le funzionalità diano il giusto risultato agli occhi di un utente esterno, viene largamente utilizzato durante le fasi pre-rilascio di un'applicazione, per verificare che i clienti non possano incontrare errori visibili. Può essere automatizzato, ma molto spesso, soprattutto nel caso di system testing, è presente anche una componente manuale, in modo da simulare in modo più accurato le azioni di una persona. Nei test a "scatola aperta", invece, si verificano dettagli strutturali e quindi di più basso livello, come il percorso del flusso di controllo all'interno del codice. Le due tipologie si mescolano spesso a formare un approccio graybox dove si cerca di testare come un utente esterno, ma considerando alcuni dei path interni del codice, questo solitamente avviene durante i primi integration test, quando il codice non è ancora totalmente sviluppato.

Performance testing

Il performance testing è un tipo di prova che permette di valutare le prestazioni di un sistema prima che vada in produzione. Al contrario delle altre categorie di test citate, non si concentra sulle funzionalità, ma su tre aspetti fondamentali:

- Velocità, cioè il grado di risposta dell'applicazione;
- Scalabilità, cioè come il programma reagisce alle variazioni nel numero degli utenti;
- Stabilità, cioè la proprietà che permette di funzionare alla stessa maniera sotto diversi carichi.

Senza performance testing l'applicazione potrebbe avere vari problemi, tra cui: risultare lenta o non funzionare quando molti utenti la utilizzano, avere inconsistenze di velocità tra i vari sistemi operativi e mostrare mancanze di usabilità. Queste caratteristiche impattano molto gli utenti, tanto che i team che ignorano o non svolgono i performance test spesso ottengono prodotti con una cattiva reputazione tra il pubblico, che porta a una diminuzione delle entrate.

Questa categoria di controlli può essere divisa in:

- Load testing
- Stress testing
- Endurance testing
- Spike testing
- Volume testing
- Scalability testing

Nei load test si verificano alcune metriche del sistema, come throughput e tempo di risposta medio, sotto i carichi di lavoro previsti, per assicurarsi che l'applicazione resti utilizzabile.

Per stress testing si intendono le prove fatte per monitorare l'andamento del sistema in condizione non usali (attacchi DoS) e quelle per controllare i limiti superiori di sopportazione del sistema.

La terza categoria riguarda i test necessari a verificare che il sistema, sotto il carico previsto, abbia prestazioni normali in modo continuo per un periodo relativamente lungo di tempo.

Lo spike testing comprova se l'applicazione riesce a gestire picchi improvvisi di carico.

I volume test, invece, sono creati per verificare le performance a fronte di variazioni nel volume dei dati gestiti dal programma, essi riguardano soprattutto i database.

Infine, l'ultimo tipo di test permette di capire se il sistema "scala" in modo corretto, in pratica si verifica che riesca a gestire continui aumenti di utenza.

3.4 Code review vs Testing

Molti team, oltre ai modelli di testing sopra citati, utilizzano la pratica della code review, cioè del "controllo del codice". Di seguito andremo a definire vantaggi, svantaggi, analogie e differenze di entrambi gli approcci.

La code review è una tecnica che permette di analizzare il codice alla ricerca di errori, codice non pulito o non elegante, mancanze dal punto di vista della sicurezza o codice troppo complesso. Lo scopo è principalmente assicurare la qualità del codice garantendo allo stesso tempo che faccia ciò che dovrebbe. La review è fatta da uno o più membri del team diversi da chi ha scritto il codice.

Il testing, che comprende tutto quello che abbiamo citato in questo capitolo, normalmente ha come unico focus quello di scovare gli errori ed eventualmente di servire da documentazione, in modo che sia più chiaro cosa una parte di codice fa, o cosa dovrebbe fare.

La revisione del codice ha vari vantaggi, primo fra tutti il fatto che lo stesso viene osservato da un'altra persona e quindi da un altro punto di vista, in secondo luogo i revisori possono far notare incongruenze tra il codice prodotto e il resto della code base (metodi con lo stesso nome, riferimenti a classi non più esistenti etc.) dando quindi una più vasta visone d'insieme. In aggiunta, se si utilizzano sia le pratiche di testing che quelle di review, queste ultime permettono di controllare la correttezza e la completezza dei test. Un altro vantaggio sta nell'apprendimento, infatti i reviewer, trovando codice non pulito, suggeriranno una maniera per migliorarlo, condividendo la loro idea con gli altri e portando ad aumento delle expertise di tutto il team.

Questa descrizione può portare a pensare che le review possano sostituire totalmente il testing, ciò invece è sconsigliato, in quanto porta a vari svantaggi che dettaglieremo di seguito. La revisione è un processo manuale e come tale è soggetto a molti errori umani e necessita di tempo, inoltre nessun reviewer sarà mai certo che un pezzo di codice non introduca side-effect o che non rompa del codice scritto in passato. In aggiunta, va specificato che il compito diventa sempre più difficile con l'aumento della quantità di codice del progetto, tanto che il push di commit in produzione (senza regression tests) si rivela molto pericoloso.

Tuttavia neanche il testing rimpiazza le code review, innanzitutto perché i test non guardano alla qualità del codice, ma anche perché il controllo di uno sviluppatore esperto può scovare errori difficilmente riscontrabili tramite le suite. Per esempio, un reviewer potrebbe accorgersi di bug che si manifestano solo durante i weekend e cioè quando i test automatici non vengono eseguiti (e.g. errata indicizzazione di un array) oppure di errori che si producono solo con un preciso numero di utenti (e.g. cattiva gestione di un caso limite).

In conclusione sia la code review che il testing hanno molti vantaggi e limiti, per cui non bisogna considerarle pratiche mutualmente esclusive, infatti molto spesso, nei team di successo, vengono utilizzate entrambe.

3.4.1 Testare i test

Un problema non banale è verificare che i test facciano ciò che devono, per validarli esistono più strategie tra cui:

- Simulare manualmente tutti i possibili casi che il test può incontrare per verificare che dia sempre la risposta corretta;
- Costruire un test automatico che verifichi il comportamento dei test;
- Effettuare delle code review sul codice di test.

Solo il primo caso dà la certezza del funzionamento dei test, ma è virtualmente impossibile da effettuare. Nel secondo metodo il problema si ripresenta ricorsivamente, e quindi quest'ultimo non rappresenta una soluzione definitiva. La terza modalità è solitamente la più utilizzata, poiché più semplice delle altre, però, per essere veramente efficace, il codice da verificare deve essere molto pulito e chiaro, in modo da evitare incomprensioni.

Capitolo 4

Sviluppo del sistema di testing

Nel seguente capitolo andrò a illustrare il modello di testing sviluppato come progetto.

4.1 Premessa

Il lavoro svolto alla Nice S.R.L, società acquisita da AWS, si è centrato particolarmente nell'automatizzare il testing di Amazon Elastic GPUs, progetto descritto nel capitolo 2. I miei compiti tuttavia sono stati vari e hanno compreso altri aspetti, come verificare il funzionamento di alcune applicazioni con EGPUs e rendere più espressivi i messaggi di errore in modo da identificarne più facilmente la causa.

4.2 Metodologia di lavoro

Tutto il lavoro è stato svolto in stretto contatto con un membro del team, che mi assegnava compiti in modo incrementale. Spesso per tracciare lo stato di avanzamento dei progetti si utilizzava una sorta di tavola kanban simile a Trello[1], erano anche frequenti meeting giornalieri, effettuati sia online che di persona. Sempre nello spirito dell'Agile si cercava di rilasciare il prodotto molto frequentemente, automatizzando più compiti possibili. La qualità del codice veniva raggiunta attraverso alcune tecniche e approcci standard, uniti con le pratiche di code review, utili anche ad individuare eventuali errori. Lo sviluppo è partito analizzando la situazione iniziale, infatti, evidenziandone problemi e mancanze, si sono potute elaborare tattiche per rendere il processo di testing più efficace. Di seguito si entrerà più in dettaglio nelle varie azioni eseguite.

[1] https://trello.com/

4.3 Situazione iniziale

La situazione iniziale del testing di EGPUs era la seguente:

- Un numero limitato di test automatici, spesso obsoleti e non perfettamente funzionanti;
- Una quantità importante di test manuali, che impiegavano una persona per quasi tutta una giornata;
- Dei test automatici pronti, ma che non venivano utilizzati perché dalla durata troppo lunga;
- Assoluta mancanza di performance test automatizzati.

I test a cui ci stiamo riferendo sono principalmente dei blackbox regression test, sia progressive che corrective. Ovviamente erano presenti anche degli unit e integration test, tuttavia, essi non rientravano nello studio richiesto.

Delineati i principali problemi si è dovuto decidere quale affrontare per primo, la scelta è ricaduta sul cercare di velocizzare i test troppo lenti in modo da poterli eseguire più spesso. I passi successivi sono stati:

- Perfezionare e correggere i test già attivi;
- Aggiungerne di nuovi;
- Includere dei performance test automatizzati.

4.3.1 Test pre-release

I test indicati nel precedente paragrafo vengono eseguiti ogniqualvolta viene effettuata una modifica al codice di EGPUs, e in particolare al driver. Così facendo si cercano di individuare eventuali errori e regressioni prima di rilasciare ai clienti il servizio. In quest'ambito si nota che la velocità dei test è fondamentale, se i test sono lenti ci si accorge in ritardo degli eventuali errori fatti, ed è molto più difficile trovarli e correggerli.

4.4 Tecnologie utilizzate

Per lo sviluppo e l'esecuzione dei test sono state utilizzate varie tecnologie, alcune delle quali proprietarie di AWS, altre di uso comune. Tra le più importanti va assolutamente citato EC2, necessario a far partire le macchine con EGPUs e testarle. Essendo i test dei blackbox, l'automazione si concentrava soprattutto nell'eseguire un programma che sfruttava EGPUs per poter successivamente raccogliere ed esaminare i risultati, per fare ciò si è usufruito pesantemente di linguaggi di scripting e regular expressions, che andremo a definire successivamente.

4.4.1 Linguaggi di scripting

Non esiste una descrizione univoca del concetto di "Linguaggio di scripting", e né di quello di "script", di seguito si darà una particolare visione, tra le più condivise, riguardo questo argomento. Per script solitamente si intende una porzione di codice che automatizza l'esecuzione di un task che, alternativamente, sarebbe eseguito manualmente da un essere umano. Gli script hanno la caratteristica di essere fortemente legati all'ambiente in cui vengono lanciati, infatti, vengono sviluppati considerando lo specifico contesto che dovranno automatizzare, che può essere un browser, un'applicazione, una shell o molto altro. La lunghezza di questi pezzi di codice può essere variabile, anche se in genere si scrivono script relativamente piccoli, alcune volte si possono raggiungere alcune migliaia di linee di codice, per cui la pulizia del codice resta importante.

Da questa definizione si può capire che sia stato necessario scrivere degli script per aumentare il grado di automazione del sistema di testing.

I linguaggi di scripting possono essere considerati quelli con cui si scrivono script, alcuni sono più portati per questo compito, ma in linea di principio tutti possono essere utilizzati per scripting. Normalmente però, si usano dei linguaggi interpretati e non compilati e che risultano particolarmente di alto livello, in modo che il codice possa essere scritto e eseguito nel minor tempo possibile. Al giorno d'oggi per fare scripting si possono utilizzare un gran numero di risorse, durante il progetto ne sono state utilizzate due in particolare, Powershell e Python.

Powershell

Powershell è un framework sviluppato da Microsoft che consiste in un linguaggio di scripting e una shell dedicata, originariamente era esclusivamente per piattaforme Windows, ma da Agosto 2016 è diventato open-source e

cross-platform. È pre-installato in quasi tutte le versioni moderne di Windows (comprese quelle server).

Il suo principale vantaggio è la presenza delle cmdlet, classi che svolgono compiti particolari, come l'interazione con il file system o i registri, in maniera piuttosto semplice e trasparente. La sintassi di Powershell, tuttavia, risulta spesso poco chiara e anche la documentazione online non è particolarmente ricca, per cui non si è ancora diffuso tra il grande pubblico.

Python

Python è un linguaggio object-oriented ad alto livello e solitamente interpretato, sebbene ci siano versioni che usano dei compilatori. Si contraddistingue per la sua sintassi particolarmente semplice, con typing e binding dinamici, che rende la scrittura del codice molto veloce. Questo linguaggio è basato su moduli e package per implementare funzioni avanzate, permettendo così di conseguire modularità e riuso. Date queste caratteristiche, Python viene utilizzato soprattutto per sviluppo veloce di applicazioni (come prototipi), per scripting e per collegare più parti di applicazioni, fungendo da "collante". Al giorno d'oggi è disponibile la versione 3, non retrocompatibile con la 2, in passato molto usata ed ancora parzialmente supportata dai creatori.

4.4.2 Regex

Una regular expression, o regex, è una stringa che definisce un pattern di ricerca con una sintassi particolare. Queste sequenze di caratteri vengono utilizzate in molte applicazioni, dai motori di ricerca agli editor di testo e il loro scopo è trovare le stringhe che fanno match con la definizione dell'espressione regolare. Rendono particolarmente veloci operazioni quali il parsing o la validazione e sono completamente indipendenti dal linguaggio utilizzato.

Sono fondamentali nell'automazione, perché possono evitare di far leggere dei file all'essere umano, o almeno ridurli considerevolmente comprimendo le informazioni più importanti. A fronte di queste caratteristiche, le regex sono state usate in modo estensivo durante il progetto, per la lettura di file di configurazione, di log intermedi e dei risultati finali, in modo che l'utente potesse essere completamente escluso dal ciclo di validazione dell'esito del test. Anche le espressioni regolari stesse sono state validate prima della loro applicazione nel codice, questo è stato fatto (manualmente) tramite l'utilizzo di regex101[2], un sito che permette di vedere l'effetto di una regex su un testo di prova.

[2]https://regex101.com/

ImageMagick

Per alcuni test è stato necessario utilizzare strumenti di gestione delle immagini, l'applicazione più utilizzata è stata sicuramente ImageMagick[3], quest'ultima permette di gestire formati grafici con semplicità, potendo applicare un gran numero di funzioni, dalla semplice rotazione al confronto pixel a pixel di due immagini. Fornisce un supporto particolarmente avanzato per lo scripting, in modo da essere utilizzabile anche senza intervento umano, per queste caratteristiche la si è introdotta nel progetto.

4.5 Testare un implementazione di OpenGL - Compliance testing

Essendo OpenGL una specifica open-source, sono note le funzionalità base che ogni sistema che dichiara di supportarlo deve avere. Per essere compliant con una certa versione quindi, si devono testare tutte le chiamate OpenGL che la suddetta versione contiene. In linea di principio ciò potrebbe essere fatto anche scrivendo test case di proprio pugno, tuttavia ciò richiederebbe assolutamente troppo tempo, per cui nel nostro caso si sono scelte due suite di test parecchio utilizzate, quella del Khronos group e Piglit.

Quest'ultime hanno degli scope diversi, in particolare i Khronos OpenGL Conformance Tests, oltre a controllare le funzioni core, verificano anche le estensioni ufficiali di OpenGL (chiamate ARB), ma non quelle multivendor (EXT) o single-vendor (chiamate con la sigla dell'azienda produttrice) affrontate invece da Piglit.

Questo genere di testing può solamente garantire che la propria implementazione contenga tutte le caratteristiche dichiarate e che queste abbiamo il corretto funzionamento base, tuttavia non sperimenta tutti i possibili casi e quindi non basta a definire il proprio codice come funzionante.

4.5.1 Khronos OpenGL Conformance Tests

Il gruppo Khronos è il manutentore di OpenGL, perciò mette a disposizione varie risorse per il testing e per verificare che le implementazioni siano compliant allo standard. In particolare fornisce del codice sorgente necessario a costruire il GLCTS test[4], che chiama varie funzioni OGL per verificare che funzionino a dovere.

[3]https://www.imagemagick.org

[4]https://github.com/KhronosGroup/VK-GL-CTS/blob/master/external/openglcts/README.md

L'eseguibile non è fornito già compilato perché il codice è scritto per essere platform-, OS-, e compiler-independent, per cui gli utenti finali dovranno effettuare il porting per poterlo lanciare nel sistema da verificare. L'avere a disposizione il codice, inoltre, è un vantaggio anche perché si possono correggere degli errori che non fanno eseguire nel modo giusto i test. Ovviamente non si può modificare tutto il codice liberamente, perché altrimenti si potrebbe influenzare l'esito delle prove, invalidandole. Dopo aver compilato l'eseguibile si possono far partire i test, ognuno può dare cinque tipi di esito:

- Ok
- Unsupported
- Fail
- Crash
- Timeout

Per poter essere compliant i risultati devono far parte solo delle prime due categorie. In caso di successo, si può procedere ad inviare dei documenti al gruppo Khronos, che, a fronte di un pagamento, fornisce la certificazione ufficiale.

Molti parametri d'esecuzione sono configurabili da command-line, alcuni permettono di selezionare quali test lanciare e altri di abilitare dei watchdog timer o di settare alcune proprietà di basso livello. Per il momento ci concentreremo su una delle caratteristiche impostabili, i pixel format descriptors (PFD).

I PFD sono delle strutture dati che definiscono alcune caratteristiche delle finestre, come la profondità di colore da usare e la possibilità di utilizzare un double buffer per le animazioni. Queste informazioni vengono utilizzate da OpenGL per eseguire i comandi. Il numero di PFD in un sistema può essere variabile e l'implementazione può essere software, a livello di OS, o hardware, a livello di scheda grafica.

I Pixel Format in un sistema possono essere molti e con caratteristiche differenti potendo produrre risultati diversi nei test, è importante quindi scegliere quali utilizzare.

In EGPUs vengono testati i PFD con queste caratteristiche:

- Supporto a OpenGL;
- Supporto all'accelerazione hardware;
- Supporto al multisampling.

che risultano essere 26, perciò il test GLCTS va eseguito 26 volte, una per ogni PFD.

Velocità test

Per ogni versione di OpenGL e per ogni PFD sono disponibili un certo numero di test (e.g. per la versione 4.3 sono 5490) che possono essere eseguiti in varie maniere:

1. Totalmente in modo sequenziale, un test alla volta e un PFD alla volta.
2. Test case sequenziali e PFD in parallelo sulla stessa macchina.
3. Test case paralleli con PFD sequenziali
4. Test case paralleli con PFD in parallelo sulla stessa macchina
5. Test sequenziali con PFD in parallelo su macchine diverse
6. Test in parallelo con PFD in parallelo su macchine diverse

Ognuna di queste modalità ha vantaggi e svantaggi, in particolare i numeri 3,4 e 6, cioè quelli con i test case svolti in parallelo, non sono accettati dal gruppo Khronos e quindi non garantiscono compliance, per cui vanno usati solo internamente per effettuare delle prove in maniera più rapida.

Tutte le altre configurazioni permettono di ottenere la certificazione, in particolare la prima è quella più usata e facile da sviluppare, tuttavia, se il numero dei PFD è alto e se la macchina e la scheda grafica non dispongono di molta potenza, può risultare molto dispendiosa in termini di tempo. Con una EGPU xlarge e una macchina di medio livello, questo tipo di test avrebbe impiegato circa 26 ore, un'ora per PFD, un tempo decisamente troppo alto. Inizialmente, come test di regressione, più che per compliance, veniva eseguito un run con un solo PFD, tuttavia l'approccio lasciava non testati tutti gli altri 25 PFD, risultando non completo.

Implementare l'approccio numero 2 è stato il mio primo compito, poiché sembrava la maniera più semplice di diminuire il tempo d'esecuzione. Far partire i run con i vari PFD in parallelo è semplice, ma porta a una serie di problemi, infatti, eseguendo 26 processi concorrenti si incappa in corse critiche e si mette a dura prova la RAM, causando degli errori nei test. Questi fallimenti risultano indistinguibili dagli altri tipi, come le regressioni, e quindi non possono essere ignorati.

Per superare questo problema si è deciso di utilizzare dei lock per garantire mutua esclusione, in particolare si è creata una mappa testName-mutex, che

appunto associa ad ogni test un oggetto di tipo Lock. In questo modo si garantisce che in qualsiasi istante di tempo τ solo il PFD π possa eseguire il test x. Quest'approccio ha vari problemi, innanzitutto istanziare una mappa con 5490 Lock è molto pesante, inoltre ciò non elimina totalmente il problema, perché due o più test computazionalmente complessi, ma con nome diverso, potrebbero essere in esecuzione allo stesso tempo, generando possibili problemi sia di RAM che di mutua esclusione, e quindi dei fallimenti.

Il secondo approccio si è basato sul mettere in mutua esclusione solo i test più pesanti e duraturi, immaginando che la causa degli errori fosse principalmente l'abuso di RAM. Per fare ciò si sono misurati i tempi di tutti i test case, per poi ordinarli in base alla durata, a questo punto si è scelta una soglia (nel nostro caso 10 secondi) per considerare il test "pesante". Successivamente si è impostato un singolo lock, che metteva in mutua esclusione tutti i test case "pesanti". In questo modo i test brevi (anche con lo stesso nome) potevano essere svolti in parallelo tra i vari PFD, mentre un qualsiasi test $x \in \Pi^5$ veniva svolto in mutua esclusione con tutti gli altri in Π, e tra i vari PFD. Questa modalità permetteva di evitare i fail, ma il lock generava delle code d'attesa tra i vari processi che innalzavano di molto il tempo d'esecuzione, infatti, un run completo con questa tecnica richiedeva circa 52 ore, un tempo addirittura doppio di quello sequenziale. Anche alzando la soglia le prestazioni non miglioravano, inoltre, per non perdere compliance, non è stato possibile cambiare l'ordine dei test per minimizzare il tempo di attesa medio, per cui anche questo approccio è stato scartato.

Non riuscendo in maniera semplice ad usare il criterio numero 2, si è deciso di passare al numero 5. Con questo metodo, avendo i vari run con i diversi PFD su macchine diverse, venivano meno tutti i problemi riguardanti RAM e concorrenza, ma la procedura di avvio e stop dei test diventava molto più complicata. Per gestire questi aspetti, sono stati sviluppati, a partire da esempi già presenti, uno script di lancio istanze e uno di terminazione delle stesse, quest'ultimi vengono eseguiti da una macchina detta launcher, che si occupa solamente di questi compiti.

L'algoritmo completo della soluzione con la modalità 5 diventa:

- Avvio launcher;
- Lettura del file di configurazione che indica il numero e il tipo delle istanze da lanciare, il numero ottimale è 26, una per ogni PFD;
- Lancio delle istanze;

[5] insieme dei test "pesanti"

- Lettura da parte delle istanze del proprio file di configurazione che specifica alcuni parametri, tra cui l'indice dell'istanza e la folder su cui caricare i risultati;
- Individuazione della versione di OpenGL supportata da EGPUs (in modo che non sia hard-coded e il test funzioni anche nelle future versioni);
- Scelta da parte delle istanze del PFD da eseguire, ciò è svolto tramite una semplice operazione di modulo basata sull'indice dell'istanza;
- Esecuzione del run;
- Memorizzazione su storage cloud del risultato;
- Download da parte del launcher di tutti i risultati delle macchine, appena essi risultano disponibili;
- Join dei risultati in un unico file;
- Caricamento del file complessivo;
- Terminazione di tutte le macchine;
- Terminazione del launcher.

Questa tattica porta ad un esecuzione corretta e compliant di tutti i test, in circa 2 ore e 30 minuti, tempo considerato accettabile seppur sia più del doppio del tempo di un singolo PFD su una singola macchina (un'ora). Questo ritardo è introdotto dalle operazioni di lancio delle macchine e dall'installazione dei software per eseguire i test, in particolare Python, il quale non è inserito di default nell'AMI[6], ma installato a runtime per garantire che ci sia sempre la versione più aggiornata. Questa metodo porta quindi a una drastica riduzione dei tempi dalle 26 ore iniziali, ma non è ancora perfetto, in quanto non gestisce in modo corretto gli errori. In particolare il problema sta nel fatto che, eseguendo tutti i test relativi a un PFD insieme, in caso di fail o crash il log generato riporta un errore generico, da cui non è possibile risalire ai singoli test in questione, per cui riprodurre e investigare il fallimento risulta molto complicato.

Per superare questa limitazione si è deciso di passare all'eseguibile di GLCTS un test alla volta, in questo modo il log diventa molto più espressivo e fine-grained, rendendo la localizzazione degli errori molto semplice. Inoltre

[6]immagine della macchina, contenente il sistema operativo e i programmi di base, è possibile creare AMI complesse a piacimento

con questa modalità è possibile gestire i crash in maniera trasparente, individuandoli e facendo partire comunque tutti i test successivi. Queste modifiche hanno avuto però una conseguenza sulle performance, aumentando il tempo d'esecuzione di circa 30 minuti, questo drawback non è stato considerato grave.

4.5.2 Piglit

Piglit[7] è un insieme open-source di test automatici per OpenGL, non forniti dal Khronos group e con uno scope particolare, cioè le estensioni EXT e quelle single-vendor, tra cui quelle sviluppate da Intel, AMD e NVidia. Per stabilire l'esito di questi test bisogna effettuare delle valutazioni particolari, infatti è ovvio che un test case che utilizza un'estensione sviluppata da Intel possa fallire su macchine AMD e viceversa, per cui bisogna innanzitutto decidere quali funzioni supportare, in modo da poter ignorare ogni altro fallimento.

Una volta deciso il subset di test da supportare si può, come con GLCTS, utilizzare il test sia per compliance sia per individuare regressioni. In questo modo l'algoritmo complessivo diventa:

- Avvio della macchina.
- Scaricamento dell'ultimo risultato del test.
- Lancio dei vari test case.
- Confronto del risultato con quello precedentemente scaricato, in caso di uguaglianza o diminuzione del numero degli errori[8] la suite viene considerata superata. In caso ci fosse un aumento dei fail, il test fallisce e vengono notificate la regressioni sul log.
- Se non ci sono regressioni la nuova soluzione viene caricata su storage cloud per essere confrontata con i successivi lanci.

Per Piglit le prestazioni non sono state un problema, infatti un run completo richiede solo 20 minuti, un tempo più che accettabile.

[7] https://piglit.freedesktop.org/

[8] possibile nel caso una nuova versione di EGPUs abbia aggiunto il supporto a delle funzioni

4.5.3 OpenGL Extensions Viewer

OpenGL Extensions Viewer[9] è un applicazione che permette di ottenere vari dettagli sull'implementazione di OpenGL, tra cui il numero delle estensioni e le diverse funzionalità supportate divise per versione. Il software contiene anche un insieme di test che permettono di visualizzare il risultato di alcuni render in base alla versione scelta, per verificarne la correttezza.

Questo programma è stato usato principalmente per verificare che EGPUs supportasse al 100% tutte le funzioni dichiarate (fino a OpenGL 4.3) tramite la schermata indicata in figura 4.1

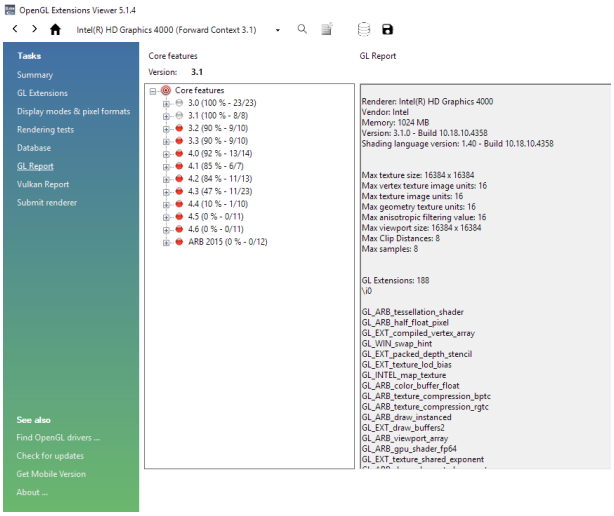

Figura 4.1: Funzioni supportate divise per versione (screenshot ricavato dalla scheda video integrata Intel, non da EGPUs)

Come gli altri compliance test, anche quest'ultimo è stato utilizzato per valutare eventuali regressioni, in particolare, per questo compito venivano utilizzati i rendering test, di cui l'output veniva confrontato (da uno sviluppatore e non ancora in modo automatico) con quello di una precedente versione.

[9] http://realtech-vr.com/admin/glview

4.6 Regression test

Come già definito nel capitolo 3, i regression test sono prove che mirano a verificare che i cambiamenti fatti al software non creino danni collaterali, inficiando il funzionamento di parti precedentemente corrette. Essendo EGPUs un servizio già avviato e rilasciato, necessita di cambiamenti al codice solo per manutenzione, che essa sia adattiva, correttiva o perfettiva, per cui le prove venivano eseguite in questo caso. In particolare i test di regressione venivano lanciati sia prima di mettere il codice in mainline che ad ogni release di una nuova versione, questo double-check è necessario perché in una versione possono entrare più modifiche diverse scritte da più persone, che singolarmente possono soddisfare i test, ma insieme possono generare errori.

In caso di individuazione di una regressione, dopo averla risolta, è prassi all'interno del team creare un test automatico in grado di simulare, in modo minimale e veloce, la situazione di errore scovata, per poter verificare che essa non si ripresenti più in futuro.

Di seguito si definiscono più in dettaglio i tipi di regression test eseguiti, ricordando però che anche i compliance test sopra citati sono stati adattati per individuare eventuali regressioni.

4.6.1 Manual test

Ad ogni release era necessario verificare che le modifiche effettuate non corrompessero il funzionamento delle applicazioni. Per fare ciò si utilizzava una lista di programmi considerati "importanti" che andavano testati. Solitamente per ogni software era presente una pagina di documentazione con indicato:

- AMI contenente il programma da testare, in modo di non doverla reinstallare a mano.
- Modalità di testing, che modelli aprire, che operazioni svolgere etc.
- Screenshot del risultato desiderato.
- Risultati nelle precedenti versioni.

Con queste informazioni lo sviluppatore poteva eseguire il testing, per poi riportare il risultato all'interno della lista, indicandone il grado di correttezza e specificando eventuali problemi di performance, crash o errori gravi. La lista andava poi confrontata con quella delle versioni precedenti e, in caso di un aumento del numero di errori, si procedeva a investigare gli stessi, verificando se si trattasse di regressioni o meno, come vedremo in 4.9

Il problema di questi test è senza dubbio la velocità, dato il grande numero delle applicazioni, per testare in modo sensato era necessario impiegare un'intera giornata di lavoro. Inoltre, la prove da effettuare non erano facilmente ripetibili, solitamente a causa di una documentazione non completa, per cui il risultato poteva essere difficile da interpretare.

Una delle mie mansioni durante il progetto sarebbe dovuta essere automatizzare un gran numero di questi test manuali, tuttavia, a causa di limiti di tempo, ciò non è stato possibile (a parte per un'applicazione, vedi 4.6.3). Il mio compito principale in quest'ambito è stato quindi eseguire le prove ad ogni versione da rilasciare e segnalare eventuali errori. Tuttavia, essendo le operazioni descritte nella documentazione molto semplici, ho cercato di esplorare più in dettaglio le applicazioni, aggiungendo nuove modalità di test e scoprendo nuovi errori, per poi curare il più possibile la documentazione per aumentare la ripetibilità e la facilità di esecuzione. Avendo informazioni più schematiche e rigorose su come eseguire i test sarà possibile, in futuro, renderli automatici in modo più semplice.

Si fa notare inoltre che, sebbene un buon numero di test manuali sia composto da pochi passi, alcuni richiedono un numero maggiore di operazioni e interventi umani, in quest'ultimo caso automatizzare risulterebbe molto costoso e non porterebbe a un gran beneficio.

4.6.2 Image-based/rendering test

In un implementazione di OpenGL come EGPUs, è fondamentale controllare la qualità del rendering per non compromettere l'esperienza utente durante l'utilizzo delle applicazioni. Per fare ciò si sono costruiti dei test che confrontano le immagini generate con un riferimento considerato corretto, questa comparazione può essere fatta visivamente da una persona o tramite strumenti automatici come ImageMagick. Durante il progetto mi sono concentrato sulla seconda categoria e ho messo mano ai test di regressione di SPECviewperf.

SPECviewperf

SPECviewperf[10] (d'ora in avanti SPEC o SP) è un applicazione di benchmarking utilizzata normalmente per valutare le prestazioni di workstation o cluster con schede grafiche. Questo software è formato da vari viewset, che sono insiemi di test ricavati da alcuni applicazioni di CAD o modellistica 3D, come Maya o Siemens NX.

[10]https://www.spec.org/

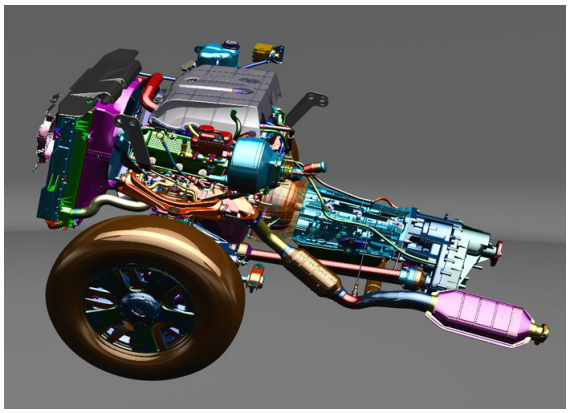

Figura 4.2: Fotogramma di un test di SPECviewperf13 relativo all'applicazione Siemens NX

Normalmente all'interno di un viewset i test vengono eseguiti in blocco, ma, per avere più controllo e log più dettagliati, ogni test può essere fatto partire anche singolarmente. Nel processo di testing di EGPUs il software viene utilizzato in maniera differente, invece di misurare le prestazioni tramite alcune metriche, come gli FPS, si confrontano le immagini generate con quelle appartenenti a una baseline. In aggiunta si fa notare che alcuni viewset non utilizzano OpenGL e quindi sono stati scartati. Il processo completo è il seguente:

- Avvio macchina di test;
- Download baseline;
- Download del file SPECXX_thresholds.json contenente le soglie di accettabilità per il confronto;
- Download file di configurazione, contenente timeout etc.;
- Download strumenti per il confronto immagini (ImageMagick);
- Esecuzione atomica di ogni test di ogni viewset, per ciascuno avvengono i seguenti passi:
  - Grab di un frame generato (per convenzione si utilizza il primo, ciò potrebbe essere cambiato modificando anche la baseline);
  - Confronto pixel a pixel del frame con quello della baseline;
  - Generazione del grado di similarità (tra 0 e 1);

  - Confronto tra il risultato ottenuto e la soglia di accettabilità relativa;
  - Scrittura del risultato del test in base alla soglia.

- Generazione del file complessivo con l'esito delle prove;
- Upload del risultato e delle immagini generate su storage cloud per debugging.

Prima dell'inizio del progetto erano presenti e automatizzati i test di SP11 e SP12, tuttavia entrambi avevano dei problemi. In particolare le soglie di accettabilità di alcuni test erano fissate a un valore troppo basso, ciò accadeva perché alcune immagini venivano generate con colori o texture casuali, causando problemi nel confronto pixel a pixel. Inoltre, dopo un attenta analisi dei log di SP12, si è notato che i test non venivano eseguiti completamente a causa di un errore di timeout, ma quest'ultimo non veniva correttamente riportato e la suite risultava passata. Il mio primo compito è stato quindi risolvere questi problemi.

Immediatamente si è capito che, sebbene il confronto pixel a pixel potesse essere appropriato per molti test, a volte risultava troppo rigido e non poteva essere l'unico algoritmo utilizzato. Per cui, aiutandosi con ImageMagick, sono stati sviluppati due nuovi algoritmi di comparazione, chiamati *structural* e *structuralColor*. Il primo algoritmo è molto semplice, ed è composto dai seguenti passi:

1. Rendere trasparenti gli sfondi dell'immagine generata e di quella della baseline;
2. Effettuare una binarizzazione delle immagini sopra citate, questo di fatto genera un immagine in bianco e nero con in evidenza solo ciò che non appartiene allo sfondo (i.e. il modello);
3. Confrontare pixel a pixel le immagini al punto due.

StructuralColor è simile a *structural*, però controlla, tra il punto 1 e 2, che il numero dei colori non resi trasparenti nell'immagine generata sia uguale al numero di colori che dovrebbe avere il modello, in pratica si verifica che il rendering non abbia alterato la quantità dei colori. È ovvio che questi algoritmi riducono le caratteristiche confrontabili di un immagine e quindi vanno usati solo nel caso il confronto pixel a pixel non sia possibile, inoltre, tra i due metodi sviluppati, il secondo è preferibile, perché più completo. Entrambi i metodi hanno un punto debole, infatti possono essere applicati solo se lo sfondo è

di un colore omogeneo e il modello non contiene al suo interno il colore del background.

La scelta dell'algoritmo va fatta in base alle caratteristiche dei singoli test, per esempio in SP11 esiste un test con un modello che può assumere randomicamente due colorazioni diverse, blu o bianco, per cui risulta sempre formato da un singolo colore ma non è dato sapere quale. In questo caso il confronto dei pixel darebbe risultati non costanti e quindi violerebbe il principio di ripetibilità dei test, generando falsi negativi in caso di cambio del colore rispetto alla baseline. Per risolvere il problema in questo caso è bastato utilizzare l'algoritmo *structuralColor* settando a 1 il numero di colori del modello. In SP12, invece, alcuni modelli vengono generati con delle texture completamente casuali e con un numero di colori non specificato, per cui è stato possibile utilizzare solo l'algoritmo *structural*.

Per rendere la scelta dell'algoritmo flessibile al massimo, è stato creato un nuovo file SPECXX_algorithms.json contenente, per ogni test, l'algoritmo da utilizzare (considerando come default non specificato il pixel a pixel) e eventuali parametri aggiuntivi, tra cui, per esempio, il colore dello sfondo dell'immagine generata. Applicando questi algoritmi ai test problematici sono stati rimossi i falsi negativi in SP11 e si sono alzate le soglie di alcuni test in SP12.

A questo punto si è passato a risolvere il problema di timeout di SP12, in pratica lo script ignorava i parametri di timeout, globali e per singolo test, passati tramite i file di configurazione e utilizzava quelli di default, troppo brevi per permettere il completamento dei test. Inoltre, quando scattava un timeout il risultato dell'ultimo test veniva ignorato, per cui, se i test precedenti erano tutti passati, SP12 ritornava un risultato positivo.

La risoluzione del problema è avvenuta per passi, in primis si sono settati i default a un valore di tempo che permettesse un esecuzione normale e senza errori, successivamente si è corretta la modalità di lettura da file per fare in modo che i parametri forniti venissero correttamente parsati, per poi gestire al meglio i casi di errore e timeout. Riguardo a quest'ultimo aspetto si è fatto in modo che, in caso di un timeout relativo a un singolo test, SPEC potesse continuare l'esecuzione, ma riportando "timeout" nella parte di log relativo al test stesso, in modo che la suite di testing segnalasse un fallimento. Nel caso di un timeout globale, invece, viene inserito nel log l'ultimo test che era in esecuzione (con risultato "timeout") e in seguito una stringa speciale che indica il timeout globale, sempre al fine di poter essere parsata dalla suite di test per segnalare l'esito corretto.

Dopo aver gestito i timeout si è pensato di aumentare la robustezza dei test, gestendo eventuali crash, dell'applicazione, del driver o della GM. Per implementare questa funzionalità si sono dapprima simulati i vari tipi di errore per osservare che exit code avrebbero prodotto, per poi inserire i controlli

appropriati all'interno del codice.

I cambiamenti fatti per SP12 sono poi stati riportati anche su SP11, per cui il processo generale è diventato:

- Avvio macchina di test;
- Download baseline;
- Download del file SPECXX_thresholds.json contenente le soglie di accettabilità per il confronto;
- Download del file SPECXX_algorithms.json contenente gli algoritmi di confronto da utilizzare con i relativi parametri;
- Download file di configurazione, contenente timeout etc.;
- Download strumenti per il confronto immagini (ImageMagick);
- Esecuzione atomica di ogni test di ogni viewset, per ciascuno avvengono i seguenti passi:
  - Grab di un frame generato (per convenzione si utilizza il primo, ciò potrebbe essere cambiato modificando anche la baseline);
  - Confronto del frame con quello della baseline in base all'algoritmo definito nel file SPECXX_algorithms.json;
  - Generazione del grado di similarità (tra 0 e 1);
  - Confronto tra il risultato ottenuto e la soglia di accettabilità relativa;
  - Controllo timeout del test;
  - Controllo timeout globale;
  - Controllo exit code;
  - Scrittura del risultato in base alla soglia e agli eventuali timeout o crash;
  - Se il timeout globale non è scattato, si prosegue, altrimenti i successivi test non vengono eseguiti.
- Generazione del file del risultato;
- Upload del risultato e delle immagini generate su storage cloud per debugging.

Con i test corretti, ci si è potuti concentrare in modo migliore sull'analisi dei risultati del confronto, mentre in SP11 tutti le immagini davano similarità dal 99% in su, in SP12 alcuni test, senza il problema dei colori sopra citato, avevano impostate soglie più basse. Il motivo di questa scelta è stato trovato analizzando le immagini generate e confrontandole a mano con quelle della baseline, in questo modo si sono notati 3 tipi di differenze:

1. Piccole differenze nei bordi dei modelli;
2. Differente gestione del padding;
3. Diverso frame renderizzazato.

Mentre la prima categoria è ignorabile e non genera grossi problemi, le altri due portano ad un abbassamento pesante della soglia di similarità. In particolare, in EGPUs alcuni test generavano immagini di dimensioni minori di quelle stabilite e, per raggiungere la risoluzione corretta, aggiungevano pixel con canale alpha = 0 che quindi venivano renderizzati come trasparenti. Nelle baseline, invece, non c'era trasparenza e il padding risultava riempito dall'ultima fila di pixel dell'immagine. Questo problema, come anche il numero 1, è originato dal fatto che la baseline è stata reperita tramite un istanza grafica autonoma, che gestisce il padding ed altri aspetti in maniera differente da EGPUs. Per risolverlo sono stati pensati due approcci, rimuovere il padding prima di confrontare le immagini tramite ImageMagick o passare a una baseline composta da immagini generate da EGPUs. La prima soluzione è sicuramente la più semplice e veloce, ma introduce un problema, se in futuro, per errore, venisse renderizzato qualcosa all'interno del padding questa regressione non verrebbe individuata correttamente. Dunque sarebbe più opportuno passare a una EGPUs-based baseline, ma per farlo servirebbe una versione di EGPUs che non abbia nessun tipo di problema di renderizzazione o confronto delle immagini, per cui le differenze citate come problema 1 dovrebbero essere considerate tutte irrilevanti e dovrebbe essere corretto anche il punto 3. Quest'ultimo errore si nota in particolari test, che, in base alla size dell'EGPUs, producono immagini diverse dalla baseline. Ciò è stato analizzato a fondo, ma non si è riuscito, nel tempo messo a disposizione, a capirne la causa.

In conclusione, le baseline di SP12 non sono ancora state cambiate, ma è stato documentato il procedimento per farlo quando tutti i problemi rimasti saranno risolti. In futuro questo porterà ad avere un set di immagini di confronto che non abbia nessun rischio di generare falsi positivi o falsi negativi.

SP11 e 12 sono software sviluppati alcuni anni fa e quindi utilizzano test di versioni obsolete delle applicazioni di modellistica, ha senso tenerli per individuare delle regressioni, ma è anche necessario introdurre qualcosa di più

moderno. Durante il progetto, quindi, si è introdotto nella suite di test SP13, la versione più aggiornata di SPEC. Il processo di scrittura del test è stato semplice, si è partiti dal codice di SP12 e lo si è attentamente analizzato e modificato per adattarlo alle caratteristiche della nuova versione. La baseline è stata generata facendo partire il benchmark sullo stesso tipo di istanza grafica autonoma utilizzata per gli altri SPEC. Non si sono riscontrati particolari problemi, a parte i tre descritti precedentemente per SP12, per cui anche in questo caso sarà necessario passare ad una baseline basata su EGPUs.

4.6.3 GUI-based testing

Per automatizzare l'utilizzo di un'applicazione ci sono due modi: usare parametri particolari attraverso la command line oppure controllare programmaticamente la GUI del software. Il primo approccio è sicuramente molto più semplice, ma molti programmi non rendono disponibile un controllo completo da linea di comando, per cui a volte ci si è dovuto affidare alla seconda modalità che però, in base alla complessità del software, può risultare dispendiosa in termini di tempo.

In EGPUs, il controllo via codice dell'interfaccia grafica è stato fatto con Pywinauto[11] una libreria Python specifica per l'automazione delle GUI, specialmente in Windows. Quest'ultima permette di avviare le finestre delle applicazioni, spostarle, selezionare delle zone, scrivere dentro i campi e avviare comandi (come per esempio la pressione del tasto Enter).

In particolare durante il progetto si è automatizzata una semplice applicazione con questa modalità, che necessitava la scrittura di un comando in una textbox e la pressione di invio. Successivamente il risultato veniva salvato dal programma in un file e di seguito parsato con semplici comandi Python e regex.

4.7 Performance testing

Oltre ai compliance e ai regression test, è stato mio compito anche mettere la base per un performance testing automatizzato, che permettesse di capire le variazioni di prestazioni da una versione all'altra. In particolare, la situazione può diventare allarmante se c'è un calo netto tra una versione e la successiva, o se delle piccole diminuzioni continue portano a un peggioramento significativo rispetto a un istanza grafica autonoma.

Come vedremo di seguito, nel tempo a disposizione si è riuscito con successo ad automatizzare la raccolta dei dati, ma non è stato possibile integrare il

[11] https://pywinauto.github.io/

codice nella suite di testing, sebbene farlo non avrebbe richiesto particolari sforzi. Inoltre, avendo iniziato da zero l'analisi, si avevano a disposizione solo un numero limitato di versioni da testare e quindi di informazioni, per cui lo studio dei risultati è stato rimandato a più avanti, quando ci sarà una mole di dati statisticamente attendibile.

4.7.1 Benchmarking

Per ottenere i dati di performance sono state utilizzate più applicazioni di benchmarking, ciascuna delle quali aveva particolari metriche per il calcolo del punteggio, le più comuni erano:

- FPS medi
- FPS massimi
- FPS minimi
- Totale frame generati
- Tempo di esecuzione

Nello specifico sono state utilizzate le tre versioni di SPEC sopra citate e due programmi della Unigine[12], Heaven e Valley. Il primo passo nello sviluppo è stato capire come automatizzare l'avvio dei benchmark. Per SPEC è bastato utilizzare gli script relativi ai regression test escludendo il confronto delle immagini, mentre, per quanto riguarda Heaven e Valley, si sono analizzati i parametri passati all'eseguibile principale durante il lancio del benchmark utilizzando il Process Explorer[13]. Una volta definito come far partire i benchmark in modo automatico, si è deciso quanti run effettuare per poter ottenere dati utilizzabili, dopo alcune analisi statistiche si è notato che dieci lanci sarebbero stati sufficienti. A seguito di ciò si sono dovuti sviluppare gli script per il parsing dei risultati, per quanto riguarda SP11, 12 e 13 è stato necessario solamente modificare la lettura dei file di log, mentre per i software Unigine è stato scritto del codice ad-hoc in Python capace di interpretare i loro output in HTML. Ottenuti dei risultati sintetici per ogni programma, un altro script li convertiva in file CSV in modo che potessero essere poi analizzati con fogli di calcolo. Per garantire facilità di accesso e disponibilità, tutti i file di output, dagli HTML ai CSV, venivano caricati tramite Python su una specifica folder nello storage cloud.

[12] https://benchmark.unigine.com/

[13] https://docs.microsoft.com/en-us/sysinternals/downloads/process-explorer

Dopo aver reso automatico tutto il processo, dal lancio al parsing, si è dovuto decidere in che modo sarebbero stati analizzati i risultati, innanzitutto era necessario un punto di riferimento per la valutazione delle prestazioni, ciò è stato trovato nei valori ottenuti lanciando i benchmark in un'istanza grafica autonoma.

A questo punto si sono improntate tre tipi di valutazioni:

- Deviazione standard dei vari test, oscillazioni troppo grandi vanno considerate come problemi di performance;
- Confronti di prestazioni tra una versione e le altre, per individuare bruschi cambiamenti, positivi o negativi, di efficienza;
- Confronti di prestazioni tra le versioni e i valori di riferimento, per controllare di essere sempre all'interno di una certa soglia rispetto alle performance dell'istanza grafica.

Relativamente a SPEC, è stato possibile estrarre i dati per ogni singolo test all'interno del programma, ottenendo una granularità più fine, mentre in Heaven e Valley i dati erano relativi ad un intero run del benchmark, risultando più ad alto livello.

Come già accennato questo lavoro rappresenta solo una base per un futuro miglioramento dei performance test, con relativa integrazione nel processo di testing ed eventuali allarmi automatici nel caso una versione non soddisfi i requisiti prestazionali.

4.8 OpenGL call recorder

Solitamente le modifiche ad EGPUs consistono nel modificare la gestione di alcune chiamate OpenGL, e dopo un cambiamento è sempre necessario far partire i regression test. Il numero di test però è alto e possono essere necessarie ore prima di avere un esito completo. Oltre a rendere più veloci i test, può essere utile schedulare i test in modo di accorgersi il prima possibile di eventuali errori. La politica di scheduling però va scelta secondo un certo criterio, uno tra i tanti possibili potrebbe essere eseguire per primi i test che utilizzano le chiamate OpenGL che hanno subito le modifiche da testare.

Per fare ciò è necessario sapere quali chiamate vengono fatte in ogni test, al fine di scoprirlo si è modificato parte del driver C++ di EGPUs. In particolare, si è iniziato aggiungendo un parametro di configurazione per EGPUs che permettesse all'utente di scegliere se effettuare o meno il logging delle chiamate. Per ogni chiamata OpenGL è stato poi necessario controllare se il flag

sopra descritto fosse settato a true, in caso positivo si procedeva ad inserire in un set un oggetto composto da:

- Il nome della chiamata
- Un array con tutti i parametri di tipo GLenum, questi sono gli unici argomenti veramente interessanti perché possono influenzare pesantemente il comportamento della chiamata, solitamente gli altri argomenti sono valori numerici destinati al disegno.

Ad ogni inserimento di un nuovo oggetto, veniva loggata in un file di debug la descrizione dell'oggetto stesso, con nome della chiamata e codici dei parametri. Si fa notare che, avendo utilizzato un set, non possono esserci oggetti ripetuti.

Come nota implementativa, specifichiamo che alcune parti dei sorgenti C++, contenenti boilerplate code considerato "necessario", non venivano scritte a mano, ma generate da uno script Python, in modo da cercare di ridurre eventuali errori di copia-incolla.

Avendo sviluppato questa estensione è diventato possibile visualizzare i tipi di chiamate effettuate da ogni singolo programma, semplicemente settando un flag di configurazione ed esaminando un particolare file di output.

4.9 Investigazione errori con EGPUs

Di seguito andremo a descrivere i vari tipi di controlli che possono essere effettuati per verificare lo stato di salute di EGPUs e per investigare gli errori nelle applicazioni. Ciò è utile a definire più in dettaglio il lavoro svolto in relazione al testing manuale delle applicazioni e alla ricerca delle regressioni.

Ogni istanza lanciata con EGPUs ha vari metodi per permettere il monitoraggio del suo stato di salute:

- Tramite la console EC2;
- Attraverso l'EGPUs manager pre-installato;
- Attraverso alcuni programmi eseguibili da prompt.

Solitamente, se si vuole la certezza che tutto stia funzionando, si esegue una combinazione di questi test. Una volta avuta la conferma del funzionamento si possono iniziare a testare le applicazioni necessarie.

Se durante il testing si incontrano degli errori bisogna innanzitutto capire come riprodurli in modo minimale, successivamente si deve scoprire se sono imputabili a EGPUs o no, questo è possibile con vari metodi:

- Disattivando il driver di EGPUs e riproducendo i passi, questo è sicuramente il metodo più veloce, ma non sempre è applicabile perché è possibile che alcune applicazioni non si aprano senza EGPUs, per mancanza dei requisiti OpenGL necessari;
- Utilizzando l'applicazione su un PC, con un implementazione OpenGL diversa da EGPUs, e non su un istanza EC2;
- Riproducendo i passi su un'istanza grafica autonoma.

Se l'errore si presenta anche in almeno uno di questi casi, esso non è imputabile ad EGPUs, e quindi non va analizzato, tuttavia è buona norma segnalarlo agli sviluppatori dell'applicazione. Invece, se l'errore si presenta solo con EGPUs, è necessario capire se si tratta di una regressione o no, questo è possibile semplicemente avviando il programma problematico su una o più macchine con versioni precedenti di EGPUs. Se l'errore incontrato non si ripresenta si ha una regressione, altrimenti no, per cui significa che si è scoperto un nuovo problema.

Le azioni da intraprendere in caso di regressioni o nuovi errori sono diverse, nel primo caso è necessario ritornare immediatamente alla versione precedente per evitare che gli utenti possano sperimentare l'errore, per poi analizzare in modo migliore il codice della modifica e scovare il bug. Invece, se il problema non è imputabile al cambio di versione, si può procedere comunque al rilascio, per poi occuparsi del problema attraverso modifiche successive.

Dopo essersi assicurati che l'errore sia dipendente da EGPUs, per individuarne la causa e risolverlo sono necessarie conoscenze particolari di informatica grafica ed OpenGL, non richieste nell'ambito del progetto, tuttavia è stato mio compito cercare di fornire il massimo numero di informazioni utili alla correzione dei problemi, e uno degli strumenti utilizzati per farlo è stato Apitrace, definito di seguito.

4.9.1 Apitrace

Apitrace[14] è un tool che permette di tracciare le chiamate alle librerie grafiche effettuate dalle applicazioni, supportando sia DirectX che OpenGL. Differentemente dall'estensione del driver citata in 4.8, quest'applicazione registra tutte le chiamate con tutti i parametri, e non solo le funzioni con signature differenti tra loro. Anche lo scopo è differente, infatti, Apitrace serve a capire che chiamate OpenGL abbiano causato un errore, questo è reso semplice anche tramite alcune funzionalità particolari, come il replay delle chiamate e il dump in un file testuale dell'ordine delle stesse.

[14]https://github.com/apitrace/apitrace

Tuttavia questo software ha alcune pesanti limitazioni, infatti durante il tracking di un'applicazione produce un rallentamento netto della stessa, che, oltre ad aumentare i tempi di testing può influenzare il comportamento del programma e nascondere eventuali errori, soprattutto nel caso di Heisenbugs[15]. Inoltre, per applicazioni non banali, sia le tracce che i log prodotti sono di dimensioni molto elevate, dal centinaio di MB fino a parecchi GB, e quindi risultano difficili da gestire.

Apitrace, per essere effettivamente utile, va utilizzato con attenzione, cercando di tracciare i casi minimali che provocano l'errore, in modo da avere in output file relativamente facili da gestire ed esaminare.

4.10 Risultati finali

In conclusione, il lavoro svolto durante il progetto ha portato ad un aumento di robustezza dei test già esistenti rendendoli più completi ed espressivi, anche attraverso un numero più elevato di informazioni nei log. Inoltre si sono aggiunti alle suite automatiche il run completo dei Khronos OpenGL Conformance Tests, il test di Piglit e quello di SPEC13, portando ad un aumento della copertura del codice rispetto agli errori. Successivamente si è lavorato ai performance test, cercando di mettere una solida base per una futura automatizzazione ed integrazione completa anche di quest'ultimi. Infine, per scopi di debugging, si è sviluppata una piccola estensione del driver di EGPUs in grado di registrare i tipi delle chiamate OpenGL effettuate dalle applicazioni.

Un lavoro completo avrebbe sicuramente necessitato di più tempo, per fare in modo di automatizzare il più alto numero di test manuali e rendere più veloce l'intero processo di testing, tuttavia ci si ritiene soddisfatti del risultato ottenuto.

[15] https://en.wikipedia.org/wiki/Heisenbug

Conclusioni

Il cloud computing al giorno d'oggi viene sempre più utilizzato dalle aziende per evitare la manutenzione di dispositivi interni, dato che il pay-per-use risulta spesso molto conveniente. Tuttavia i fornitori di servizi non riescono sempre a garantire la flessibilità necessaria a competere con soluzioni custom, per cui stanno andando verso soluzioni sempre più modulari, che possano dare libertà di scelta al cliente.

Servizi di questo genere, come Elastic GPUs, pur rimanendo relativamente semplici da usare per un utente, risultano particolarmente complessi da costruire, per cui è difficile anche verificare il loro funzionamento. Dalla nascita dell'Agile e del Test-Driven Development il tema del testing ha iniziato a riscuotere sempre più successo e a non essere considerato una perdita di tempo, portando così allo sviluppo di tecniche sempre più avanzate in quest'ambito. Tuttavia, le verifiche di un software, per quanto possano essere progettate ad alto livello, dipendono sempre il larga parte dall'ambiente e dal contesto del progetto e quindi non c'è mai una maniera univoca di implementarle.

Durante il progetto di tesi si è dovuto infatti studiare l'ecosistema di OpenGL e l'architettura di Amazon Elastic GPUs, per poi decidere quali tipi di test fossero appropriati, scelta poi ricaduta sulle categorie di compliance, regression e performance.

L'obiettivo del mio progetto è stato quello di rendere automatici quanti più test possibile, cercando inoltre di aumentare la velocità di quelli già sviluppati e rendendo espressivi al massimo i messaggi di errore, in modo da capire semplicemente e rapidamente la causa di eventuali problemi. Credo che lo scopo sia stato in buona parte raggiunto, sebbene, avendo più tempo, si sarebbero potuti migliorare alcuni aspetti.

In conclusione questa tesi ha cercato di introdurre i concetti relativi al cloud computing ed in particolare ad Amazon Elastic GPUs, dando poi una visione sulle tecniche di testing utilizzate al giorno d'oggi ed infine mostrando le strategie di verifica effettivamente implementate.

Sviluppi futuri

L'automatizzazione del testing di Elastic GPUs non è stata completata al 100%, infatti si sarebbe dovuta dedicare più attenzione ai cosiddetti manual test, prove che consistevano nell'utilizzo di una particolare applicazione da parte di uno sviluppatore, in modo da verificare se tutte le caratteristiche del programma fossero funzionanti come previsto. Si fa notare però che, data la complessità di alcuni software, ciò avrebbe potuto richiedere molto tempo.

Inoltre, per quanto riguarda i performance test, si è posta solamente una base per un futuro sviluppo, che potrebbe comprendere analisi statistiche automatiche e allarmi generati in caso di calo repentino di prestazioni.

Ringraziamenti

Vorrei ringraziare il Prof. Andrea Omicini per avermi permesso di affrontare questo progetto e il Dott. Fabio Lagalla per il supporto e la disponibilità forniti durante tutto il tirocinio e il lavoro di tesi.

Un ringraziamento speciale va a tutta la mia famiglia che mi ha continuamente supportato durante i cinque anni di studio e le varie disavventure in giro per l'Italia e l'Europa. Ci tengo a ringraziare anche tutti i miei amici di Castelferretti, in particolare: Chiara F., Chiara G., Edoardo, Giulia, Jacopo, Jin, Maddalena, Michele, Nicola, Riccardo, Samuele B., Samuele S. e Tommaso, che pur sparsi per il mondo, sono sempre presenti quando necessario.

Un altro gruppo fondamentale al conseguimento di questo titolo è stato quello formato da tutti gli amici/colleghi informatici del corso tra cui: Alberto G., Alberto M., Alessandro, Alessio, Chiara, Christian, Emanuele, Enrico, Lisa, Marco, Matteo, Michele, Simone e Stefano, con cui sono state condivise le gioie e i dolori della programmazione. È necessario che io ringrazi anche Martina, che, non essendo stata citata nei ringraziamenti triennali, si era arrabbiata. Una delle persone che non dimenticherò mai è sicuramente Mascia, che mi ha nutrito per cinque anni sempre col sorriso in faccia. Vanno citati anche tutti i miei coinquilini avuti a Cesena nel corso di questo cammino universitario: Daniele, Davide, Eugene, Francesco, Giansalvo, Giulia, Giulio, Leonardo, Luca, Max, Nicola, Roberto, Simone e Vincenzo.

Tengo que darle las gracias a toda la gente conocida en Valencia que me hice vivir una gran experiencia, empezando por mi compañeros de piso, Adriano, Alba, Stanislav e Víctor, pero sin olvidar el grupo "VIP", en particular: Aidée, Catherine, Daniele, Enrico, Federica, Irene, Jacopo, Julie, Karina, Martin, Rossella, Sara, Sid, Stefano e Tommaso.

Ultime, ma non ultime, vanno ringraziate tutte le persone conosciute ad Asti, i miei coinquilini Gabriele e Mattia, che mi hanno permesso di apprezzare la città, e tutti i colleghi della Nice, che mi hanno accolto nella loro azienda in maniera veramente calorosa.

Bibliografia

[1] Stallings, William, et al., *Computer security: principles and practice*, Pearson Education, 2012.

[2] Cloud Security Alliance, *Top Threats to Cloud Computing V1.0*, CSA Report, 2010.

[3] Martin, Robert C. *Clean code: a handbook of agile software craftsmanship*, Pearson Education, 2009.

[4] Myers, Glenford J., Corey Sandler, and Tom Badgett. *The art of software testing.* John Wiley & Sons, 2011.

[5] Pettichord, Bret. *Four schools of software testing.* 2003.

[6] Leung, Hareton KN, and Lee White. *Insights into regression testing (software testing).* Software Maintenance, 1989., Proceedings., Conference on. IEEE, 1989.

[7] Munshi, Aaftab. *The opencl specification.* Hot Chips 21 Symposium (HCS), 2009 IEEE. IEEE, 2009.

[8] Buckland, Mat. *Programming Game Al by Example.* Jones & Bartlett Learning, 2005.

Sitografia

[9] Andrea Omicini, *Computing without space - Cloud Computing*, http://campus.unibo.it/id/eprint/316585, Dicembre 2017.

[10] AWS, *Amazon Web Services official site*, https://aws.amazon.com

[11] Viroli Mirko, Casadei Roberto, *Testing*, http://campus.unibo.it/id/eprint/334999, Maggio 2018.

[12] Shivani Thakar, *Importance Of Software Testing*, https://www.metaltoad.com/blog/importance-software-testing-0

[13] Codacy, *Code Review vs. Testing*, https://www.codacy.com/blog/code-review-vs-testing/

[14] David Gilbertson, *Code reviews vs. unit tests — fight!*, https://hackernoon.com/code-reviews-vs-unit-tests-fight-bea122b37614

[15] Riccardo Franconi, *User stories: una guida pratica*, https://www.ideato.it/technical-articles/user-stories-una-guida-pratica

[16] Guru99, *Guru99 official website*, https://www.guru99.com

[17] AWS, *Delivering Powerful Graphics-Intensive Applications from the AWS Cloud*, https://www.slideshare.net/AmazonWebServices/new-launch-delivering-powerful-graphicsintensive-applications-from-the-aws-cloud

[18] AWS, *CMP208_Unleash Your Graphics Solutions with the Flexibility of Elastic GPUs*, https://www.slideshare.net/AmazonWebServices/cmp208unleash-your-graphics-solutions-with-the-flexibility-of-elastic-gpus

[19] AWS, *Deep Dive on Amazon EC2 Elastic GPUs - May 2017 AWS Online Tech Talks*,

https://www.slideshare.net/AmazonWebServices/deep-dive-on-amazon-ec2-elastic-gpus-may-2017-aws-online-tech-talks

[20] Dmitry Guzeev, *What is OpenGL and how to start learning it?* https://medium.com/@wrongway4you/what-is-opengl-and-how-to-start-learning-it-34f19cfa219f

[21] Wikipedia, *Wikipedia official site*, https://en.wikipedia.org

[22] Miguel Angel Sepúlveda, *What is OpenGL?* http://www.linuxfocus.org/Portugues/January1998/article15.html

[23] Samsung, *What Is a Graphics API?* https://developer.samsung.com/tech-insights/vulkan/what-is-a-graphics-api

[24] TurboFuture, *What Is DirectX?*, https://turbofuture.com/computers/What-Is-DirectX-How-Does-DirectX-Work

[25] StreamHPC, *What is OpenCL?*, https://streamhpc.com/knowledge/what-is/opencl/

[26] AMD, *Episode 1: What is OpenCL™*, https://www.youtube.com/watch?v=aKtpZuokeEk

[27] StackOverflow, *StackOverflow official website*, https://stackoverflow.com

[28] Fallout Software, *OpenGL Starting Point Tutorial*, http://www.falloutsoftware.com/tutorials/gl/gl2.htm

[29] Python Software Foundation, *Python official website*, https://www.python.org

[30] Jonny Fox, *Regex tutorial — A quick cheatsheet by examples*, https://medium.com/factory-mind/regex-tutorial-a-simple-cheatsheet-by-examples-649dc1c3f285

[31] Microsoft, *Writing HLSL Shaders in Direct3D 9*, https://docs.microsoft.com/en-us/windows/desktop/direct3dhlsl/dx-graphics-hlsl-writing-shaders-9

[32] Microsoft, *DirectCompute*, https://blogs.msdn.microsoft.com/chuckw/2010/07/14/directcompute/