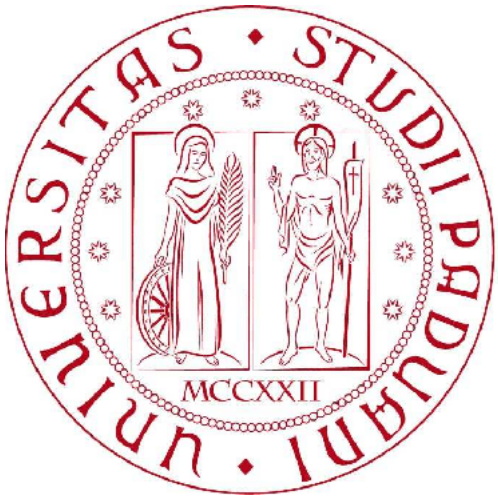


UNIVERSITÀ DEGLI STUDI DI PADOVA
Dipartimento di Ingegneria
dell'Informazione
Corso di Laurea Triennale in Ingegneria Informatica

TESI DI LAUREA

Android NDK e compilatore Just-In-Time: prestazioni a confronto

RELATORE: Carlo Fantozzi
LAUREANDO: Marco Mazzucato

Anno Accademico 2012 - 2013

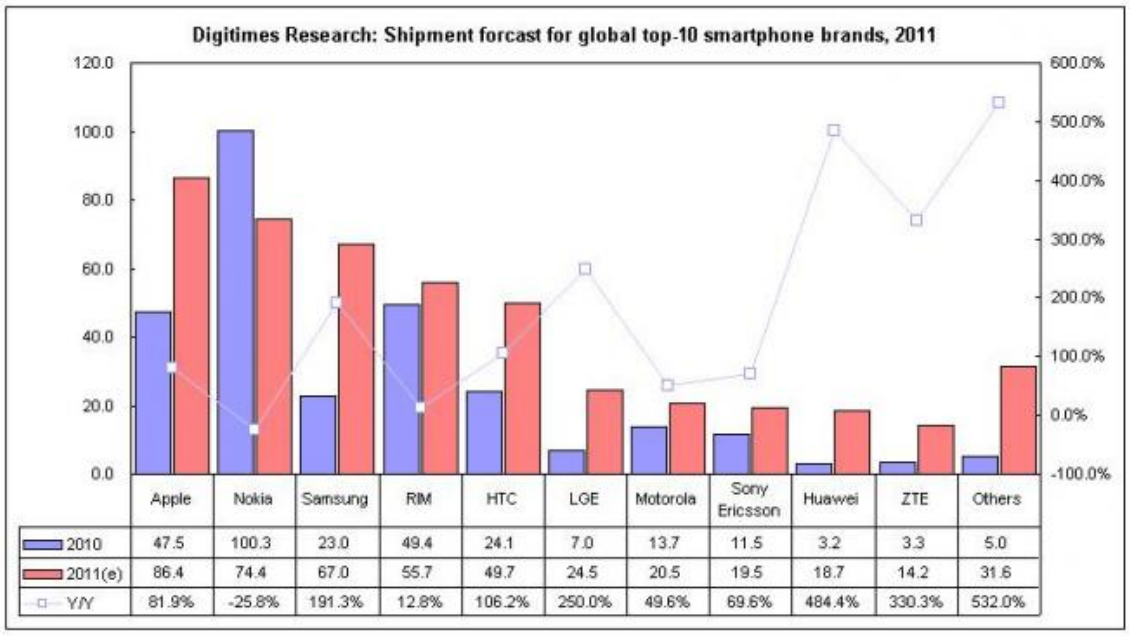
Sommario

Questa tesi è volta all'analisi degli strumenti NDK (sigla di *Native Development Kit*), ossia strumenti complementari al normale sviluppo per applicazioni Android, studiati ed introdotti in ottica prestazionale.
Oltre ad una presentazione formale su tale pacchetto ed un'analisi sui requisiti richiesti affinché le applicazioni abbiano rilevanti benefici direttamente tratta dalla documentazione ufficiale ed ad alcuni cenni sugli strumenti utilizzati, è stata svolta un'analisi il più possibile accurata misurando effettivamente la differenza di prestazioni in due algoritmi. Tale esperimento si è svolto sviluppando due modelli di applicazioni: da un lato programmi in puro Java per Android, dall'altro programmi Java con alcuni metodi in C/C++, ossia in *linguaggio nativo*, inseriti tramite le librerie NDK.
In particolare sono stati confrontati i tempi rilevati dei metodi analoghi e sono stati ricavati grafici temporali in rapporto alle dimensioni dei dati in ingresso.

Indice

Introduzione

Il settore dell'Informatica è la dimostrazione probabilmente più emblematica del dinamismo tecnologico che ha coinvolto e coinvolge l'essere umano. Oggi è un settore che non conosce crisi, anzi, con l'avvento degli smartphone il suo sviluppo è addirittura cresciuto, ampliandosi quindi in modalità differenti dai "classici" personal computer. Le funzioni che pochi anni fa richiedevano un computer ormai sono completamente assolte da questi nuovi dispositivi che stanno godendo di una diffusione a macchia d'olio a livello mondiale, accompagnata da una crescita di potenza di calcolo e di prestazioni senza paragoni anche rispetto ai personal computer. Basti pensare che uno degli smartphone che ha avuto più successo fino ad oggi, il Samsung Galaxy S, vanta un processore da 1GHz, mentre l'HTC X One monta un processore Tegra quad core, passando quindi da poco più che semplici cellulari a veri e propri calcolatori tascabili. Nonostante la sopra citata grandissima diffusione, le piattaforme presenti sul mercato sono principalmente soltanto tre: Android, iOS e Windows Phone.
Ad oggi la piattaforma smartphone più diffusa sul mercato è certamente quella Android, un sistema basato su kernel Linux compatibile con una grande varietà di hardware, sviluppato in gran parte da Google Inc. L'estrema flessibilità, i sorgenti open-source della totalità del sistema hanno portato una grande varietà di produttori (Samsung, Motorola, LG, HTC, Acer, Sony, Toshiba, Huawei,...) ad utilizzare questo sistema come propria piattaforma, integrandolo molto spesso con proprie applicazioni e portando il numero di dispositivi Android venduti al tasso di crescita di oltre il 600% dal 2008.



	Apple	Nokia	Samsung	RIM	HTC	LGE	Motorola	Sony Ericsson	Huawei	ZTE	Others
2010	47.5	100.3	23.0	49.4	24.1	7.0	13.7	11.5	3.2	3.3	5.0
2011(e)	86.4	74.4	67.0	55.7	49.7	24.5	20.5	19.5	18.7	14.2	31.6
Y/Y	81.9%	-25.8%	191.3%	12.8%	106.2%	250.0%	49.6%	69.6%	484.4%	330.3%	532.0%

Figura 1: Vendita di smartphone delle principali 10 case produttrici tra il 2010 e il 2011. Da notare che soltanto Nokia ed Apple non hanno sistema Android, oltre ad alcuni sporadici modelli Samsung, con il sistema Bada, e HTC, con Windows Phone.

Capitolo 1

Android, SDK e NDK

1.1 La struttura di Android

Come già accennato nell'*Introduzione* Android è un sistema sviluppato basandosi su kernel Linux che fornisce nativamente i driver hardware per la gestione delle periferiche, dal Bluetooth, all'ingresso/uscita USB, al display.
Secondo un layer rigido, sono state integrate le librerie di sistema, tra cui il DBMS (DataBase Management System) SQLite e quella grafica OpenGL/ES che forniscono metodi globali di supporto all'intero sistema di applicazioni. A questo livello si può ricondurre anche la Java Virtual Machine denominata Dalvik che supporta ed esegue le applicazioni caricate ed in stato *awake*.
Al livello successivo vi sono i framework delle applicazioni basilari, come il sistema di gestione delle finestre, il gestore di pacchetti ed il manager della parte telefonica (che mette a disposizione elementi basilari per la gestione delle chiamate in ingresso ed in uscita), ossia parti proprie di Android stesso, non più quindi di Linux generico.
All'ultimo livello, come è facile intuire, vi sono le applicazioni (o *app*, termine introdotto per la prima volta da Steve Jobs, fondatore della Apple) installate dall'utente, da Google (Maps, Navigatore, Play Store, Search, Gmail, Calendar, Talk,...) o dall'azienda sviluppatrice del dispostivo mobile, che possono andare da semplici tastiere software a gestori multimediali interfacciabili con dispositivi esterni a gestori di social network.
Da non dimenticare che alcuni dispositivi l'interfaccia utente appare modificata anche in modo drastico, integrando sotto-sistemi proprietari, ad esempio *HTC Sense* e il *Motoblur* della Motorola per le più svariate finalità, tra cui fornire widget per accesso diretto ai social network.
Un'altra caratteristica tecnica di Android è come il sistema sia sviluppato in ROM firmware, cioè archivi di file preparati appositamente per tale dispositivo, quindi non in pacchetti universalmente compatibili. Se da un lato questo può apparire strano, visto che i driver sono sempre praticamente gli stessi, da un lato consente di ottenere

pacchetti molto leggeri da importare nella scheda di memoria e gestire con programmi appositi per installare in qualche minuto un nuovo sistema Android. Intere comunità (come XDA) si dedicano allo sviluppo di ROM sviluppate e personalizzate per praticamente ogni tipo di smartphone, generalmente per fornire versione nuove di Android a dispositivi "abbandonati" dai produttori. Resta da considerare che per l'installazione manuale di un sistema Android su un dispositivo bisogna ottenere i permessi di superutente Unix ossia di *root*, opzione normalmente inibita per motivi tipicamente commerciali oltre che legali, visti i programmi di overclocking e di gestione delle risorse che con tali privilegi si possono installare (ad esempio esistono programmi per escludere l'accesso ad un router wireless a qualunque altro dispositivo). Inoltre cambiare sistema in questo modo comporta la decadenza della garanzia del dispositivo, cosa che ovviamente non avviene aggiornando tramite software ufficiale il telefono.

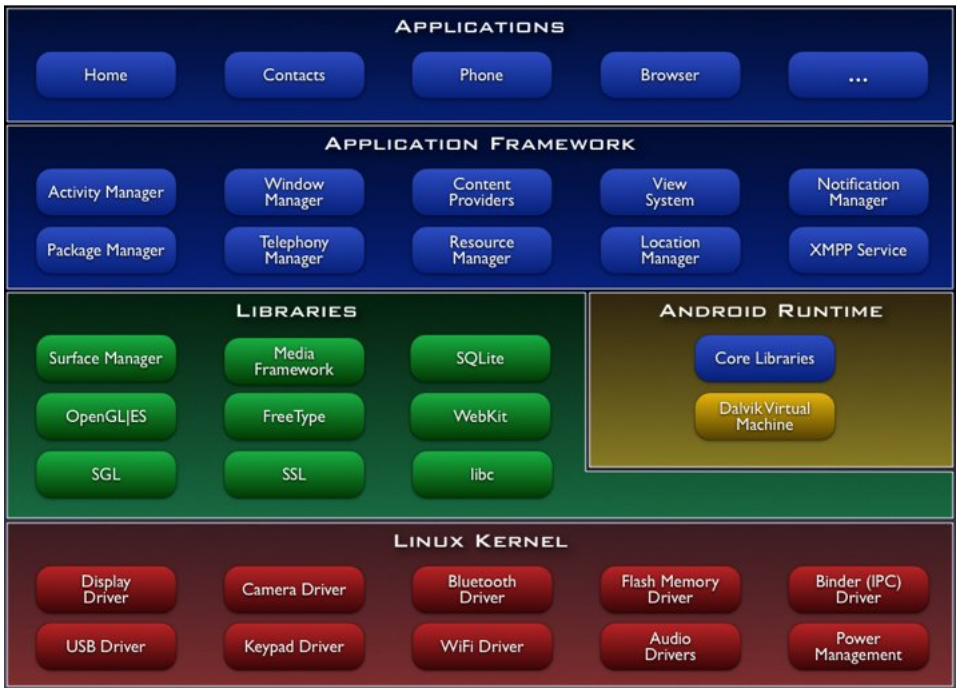


Figura 1.1: Architettura a blocchi indipendenti del sistema Android. Tale struttura facilita la programmazione finale, in quanto non è necessario occuparsi delle strutture di basso livello.

1.2 Flessibilità e semplicità di sviluppo

Gli aspetti che davvero contraddistinguono la piattaforma Android sono la flessibilità e la portabilità, che la rendono adatta a funzionare su diverse architetture (non solo su smartphone e tablet, ma anche su sistemi x86, come i personal computer), a controllare in modo personalizzabile le risorse a disposizione adattandosi a tutti gli usi possibili ed a godere di un market, il Play Store, su modello di quello Apple. Tuttavia gli aspetti più originali si respirano davvero solo programmando applicazioni per questo sistema:

infatti lo sviluppo è basato su delle librerie open-source, le SDK (*Software Development Kit*), che permettono utilizzando Java, non linguaggi "proprietari" come C# o Objective-C, di costruire applicazioni in modo semplice, specialmente integrando alla programmazione l'utilizzo degli strumenti Eclipse, un ambiente di sviluppo open-source che dal 2008 collabora con Google fornendo supporto per l'installazione di plug-in volti esclusivamente alla programmazione Android.
L'estrema semplicità e la gratuità di tali strumenti ha portato il market Android a vivere una vera e propria esplosione di applicazioni per i più svariati scopi, che talvolta richiedono prestazioni al di là del dispositivo o che vengono limitate da Java, che si presenta come un linguaggio ad alto livello, estremamente portabile, robusto, fortemente tipizzato e semplice nella sua logica, tutti aspetti che purtroppo costano in termini di velocità e di prestazioni. L'esempio più lampante e più famoso di questo linguaggio è il *Garbage Collector*, uno strumento invocato in modo non prevedibile dalla macchina virtuale per ripulire l'heap dell'applicazione da strutture di dati ed oggetti non più utilizzabili, rendendo più pulita la programmazione, ma generalmente anche molto più lenta l'esecuzione del programma stesso.

1.3 NDK: le prestazioni prima di tutto

Applicazioni lente non sono ovviamente tollerate sui PC, ed in dispositivi portatili sono ancor più detestate, in quanto dagli smartphone si pretendono risposte immediate. Non ha effettivamente senso attendere svariati secondi per visualizzare gli sms ricevuti o qualche minuto per l'apertura di un'applicazione che abbia una grafica d'alto livello. Quindi, se da un lato i dispositivi esibiscono sempre più prestazioni "spinte" (ad esempio l'HTC One X monta un processore Tegra 3 quad core), dall'altro servono tecniche per sviluppare programmi in grado di fornire prestazioni sempre migliori, adottando talvolta piccoli accorgimenti negli algoritmi fino ad arrivare a ristrutturare totalmente il codice. La prima tecnica è programmare la parte grafica, se essa costituisce un aspetto rilevante, tramite librerie OpenGL le cui istruzioni possono essere elaborate dalla GPU del dispositivo mentre la CPU elabora il core del programma, in pratica *parallelizzando* il lavoro del dispositivo, per quanto ovviamente possibile. Questa tecnica generalmente è utilizzata in giochi che costituiscono una fetta sempre più rilevante del mercato. Da non trascurare il fatto che sfruttare CPU e GPU in parallelo per molto tempo comporta un elevato consumo della batteria, indipendentemente dalla qualità della macchina.
Tuttavia spesso si riscontrano altre esigenze totalmente differenti dalla grafica: è possibile che il programma necessiti di un gran numero di operazioni che non richiedono l'interazione con l'utente, ad esempio un pattern matching su un testo, operazioni per cui il Java risulta poco performante perché è un linguaggio interpretato. Per ovvia-

re al problema è stato rilasciato praticamente comporaneamente al primo SDK un pacchetto aggiuntivo opzionale, l'NDK (*Native Development Kit*), che consente la programmazione di porzioni di codice in linguaggio "nativo" usando C, C++ ed Assembly per processori ARM, linguaggi per definizione orientati alle prestazioni che tuttavia presentano un livello di difficoltà superiore rispetto al Java, offrendo una varietà di costrutti differenti e talvolta più complessi da utilizzare, oltre ad una tipizzazione nettamente inferiore.
Come è riportato sulla documentazione ufficiale [1], non bisogna credere che l'NDK sia uno strumento magico che aumenterà sempre le prestazioni dell'applicazione mentre ne aumenterà sempre la complessità, sia di programmazione che di compilazione, richiedendo più competenze tecniche della comune programmazione Android. L'utilizzo "dovrebbe essere limitato a porzioni di codice che eseguono operazioni di notevole intensità per la CPU", anche se tramite l'NDK è possibile riciclare codice già scritto in linguaggio nativo senza troppe modifiche.
In questa tesi l'utilizzo degli strumenti NDK sarà volto al confronto di prestazioni tra metodi scritti in Java e quelli scritti in C++ utilizzati su programmi che svolgono operazioni certamente orientate al calcolo puro, come il prodotto canonico di due matrici e l'ordinamento tramite quicksort di un array, ossia programmi che non richiedono interazioni con l'utente, in quanto in tal caso sarebbe palesemente inutile utilizzare linguaggi più prestazionali: il tempo di reazione dei riflessi naturali umani è nell'ordine dei centesimi di secondo, mentre la frequenza di clock dei processori in questione è nell'ordine dei Gigahertz, con tempi quindi nell'ordine dei nanosecondi.

1.4 Struttura dell'NDK

La struttura ed il funzionamento dell'NDK sono estremamente semplici e risalgono alla JNI (*Java Native Interface*) ossia un modo introdotto per utilizzare con applicazioni Java del codice nativo, progettato inizialmente per riutilizzare il codice già esistente in C, che storicamente è un linguaggio più datato di Java: basti pensare che C risale al 1972 quando Dennis Ritchie lo progettò mentre Java è molto più giovane, infatti risale al 1992 anche se fu annunciato ufficialmente solo tre anni più tardi dalla Sun Microsystems.
L'interfacciamento tra i due linguaggi è basato sulla definizione e sull'introduzione di un insieme di classi di raccordo fra i due contesti, che presentano un'interfaccia Java, ma che delegano al codice nativo l'implementazione dei metodi. Lo stesso framework consente anche l'operazione inversa, ovvero l'invocazione di codice Java da parte di programmi scritti nei linguaggi nativi, funzione raramente utilizzata al punto tale da essere quasi dimenticata, in quanto sarebbe una scelta concettuale che comporterebbe codice più complesso e più lento, per cui un'opzione piuttosto deprecabile.

La struttura del progetto diventa inevitabilmente più complessa, vista l'aggiunta forzata di un file che contiene le funzioni implementate in linguaggio nativo e di un file di configurazione, ossia un file .mk anche chiamato *makefile*, che indica alcuni dei parametri fondamentali del compilatore C `gcc` senza i quali il programma non verrebbe interpretato in modo corretto. Nel dettaglio tale file contiene il nome del modulo del programma, che di default coincide con il nome del progetto, il nome di tutti i file che contengono metodi implementati in linguaggio nativo e che quindi devono essere inclusi, i flag di compilazione (come il grado di ottimizzazione) e le librerie aggiuntive dell'NDK da importare. Tali librerie originariamente non erano previste nella JNI, ossia sono state prettamente sviluppate per Android, come la libreria `llog` che, come indica il nome, serve a stampare alcuni messaggi sul log di Android da linguaggio nativo, utilizzata nei programmi per controllare il corretto avanzamento.
Le seguenti righe riportano il contenuto del file Android.mk del progetto che svolge il QuickSort.

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := QuickSortC
LOCAL_SRC_FILES := QuickSortC.cpp
LOCAL_LDLIBS := -llog
LOCAL_CFLAGS += -O3
include $(BUILD_SHARED_LIBRARY)
```

Quasi tutte le direttive al compilatore sono auto-generate dall'IDE utilizzato e comunque di natura standard, tra cui il percorso locale (`LOCAL_PATH`), la pulizia delle variabili d'ambiente, i moduli locali e i relativi percorsi (rappresentati dalle variabili `LOCAL_MODULE` e `LOCAL_SCR_FILES`), per concludere con l'inclusione della direttiva per costruire le librerie condivise tra la JVM e la JNI.
Le restanti due righe indicano che per una corretta esecuzione serve l'inclusione della libreria che coordina il log di sistema
`LOCAL_LDLIBS := -llog`
e che il codice deve essere ottimizzato con ottimizzazione di grado 3
`LOCAL_CFLAGS += -O3`.

L'aspetto più delicato e l'unico ancora da affrontare è il passaggio di parametri tra i due "mondi", tra Java e C/C++. JNI e NDK forniscono un modo per appianare le differenze sostenziali tra i due tipi di linguaggio con una sintassi sviluppata ad-hoc: i tipi fondamentali conservano il nome originale, preceduti da una 'j' che simboleggia la provenienza di quella variabile dall'ambiente Java, mentre gli array sono trattati come oggetti dell'opportuna classe `j<tipo>Array` vista la netta differenza tra C e Java relativamente agli array. Infatti i puntatori presenti nel C permettono l'allocazione di liste di elementi semplicemente allocando un'area di memoria, quindi prevedendo

metodi per l'allocazione sia statica che dinamica, mentre il Java, concepito per essere più rigido e strutturato oltre ad essere più tipizzato, gestisce gli array come oggetti particolari, all'interno dei quali vi sono liste preesistenti. A prova di tale aspetto si può considerare che ogni array di Java contiene al suo interno la variabile di istanza *length*, indicante il numero degli elementi presenti nello stesso, abbandonando quindi il concetto di una semplice lista di elementi tutti dello stesso tipo.

```
int arrayJava[] = new int[100];
```

```
int *arrayC = (int *) malloc(100 * sizeof(int));
```

Le due righe di codice mostrano l'allocazione di due array, il primo in Java e l'altro in C. Da notare l'utilizzo della parola chiave `new` nella prima riga, in netto contrasto con la chiamata ad una funzione delle librerie come la `malloc` nella seconda. Inoltre si evidenzia l'utilizzo dei puntantori del C, oltre alla dimensione non costante (legata all'architettura) delle variabili primitive, motivo per l'utilizzo del comando `sizeof`.

La seguente tabella riporta alcune delle variabili primitive più importanti, con il confronto sui tipi di variabili, evidenziando varie diversità tra Java e C, che il framework NDK (o meglio JNI) risolve con l'introduzione di modelli di variabili costruiti ad-hoc allinenando di fatto il C al modello delle variabili Java, in quanto quest'ultimo, oltre ad offrire un corredo più completo, ha una struttura di dimensione standard al contrario dell'ambiente nativo nel quale la dimensione delle variabili è determinata dalle word dell'architettura (che quindi possono essere 8, 16, 32, 64 o 128 bit).

Tipo Nativo	**Tipo JNI**	**Descrizione**
unsigned char	jboolean	unsigned 8 bits
signed char	jbyte	signed 8 bits
unsigned short	jchar	unsigned 16 bits
short	jshort	signed 16 bits
long	jint	signed 32 bits
long long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

Tabella 1.1: Confronto tra i nomi delle variabili in C e in JNI

Il prossimo paragrafo approfondisce le tecniche e gli aspetti più tecnici della comunicazione tra i differenti ambienti.

1.5 La comunicazione tra ambienti

Si deve tenere conto che i due ambienti, Java e quello nativo, devono ovviamente comunicare in modo trasparente, cosa resa possibile dalle tecniche di interfaccia che fornisce la JNI. Come in ogni trattazione riguardante dati da scambiare, occorre distinguere tra parametri passati per *valore* e quelli passati per *riferimento*.
Per i primi, valgono le regole elementari di Java: il principio di base è che il dato, praticamente sempre e solo di tipo fondamentale, è di dimensioni ridotte e quindi viene effettuata una copia run-time dello stesso e passata alla funzione che lo richiede. In pratica si può operare su tali dati senza influenzare gli stessi alla sorgente.
Per i dati passati per riferimento il meccanismo è opposto: sono dati non elementari, tipicamente oggetti o strutture che, evitando di replicare elementi di dimensioni potenzialmente elevate, vengono passate associando un riferimento all'oggetto. Di conseguenza, una modifica all'interno del metodo si ripercuote (generalmente) sulla variabile iniziale. È intuitivo che il problema principale verta su queste operazioni: di fatto si deve passare un riferimento di un oggetto di Java *all'esterno* della JVM, ossia al di fuori dell'ambiente Java.
Tale passaggio di strutture dati viene implementato tramite una variabile d'ambiente chiamata nella riga di codice `env`. Questa variabile altro non è che un puntatore ad un'area di memoria nella quale vengono salvate le strutture richieste per l'esecuzione della funzione in questione. Pertanto ogni metodo nativo che vorrà utilizzare le suddette variabili dovrà prelevarle sfruttando il puntatore `env`.
Le seguenti righe d'esempio illustano come accedere ad un array, chiamato`array` tramite la variabile `env`.

```
jintArray a = env->GetIntArrayElements(array, JNI_FALSE);
```

Da notare l'utilizzo della funzione apposita `GetIntArrayElements` che coordina anche l'eventuale copia dei dati in ingresso: è possibile lavorare su una copia dell'array in modo che ogni modifica non alteri i dati iniziali. La variabile `JNI_FALSE`, che corrisponde a 0, indica che l'array non viene copiato, quindi si lavorerà direttamente sui dati salvati in memoria [2, Array Operations].
Un altro elemento, che apparirà scontato ma che in fase di programmazione spesso non si rivela tale, è il riconoscimento corretto dei metodi: affinché un programma Java possa utilizzare una funzione scritta in C/C++ bisogna legare il prototipo dichiarato in Java con il metodo nativo relativo alla sua implementazione nel file `.c` o `.cpp`, onde evitare errori in fase di runtime per l'effettiva mancanza del metodo dichiarato in Java. Quindi la classe Java dovrà caricare la libreria, o il modulo, coordinato dal `makefile`, mentre il nome funzione associata dichiarata nel file nativo dovrà contentere il percorso del package preceduto da "Java", inoltre dovrà essere dichiarata esterna specificando il tipo di linguaggio utilizzato, ad esempio "C" per C e C++. Un eventuale errore nel-

l'impostazione porterà il programma alla chiusura in esecuzione e non in compilazione, in quanto l'interprete non è in grado di comprendere a priori l'assenza del metodo nativo.
In pratica il metodo che nella classe Java viene semplicemente identificato con la propria firma (nome del metodo e lista dei suoi parametri), nell'ambiente JNI prende il nome

`Java_package_nomeClasse_nomeMetodo`

Ovviamente il nome non è l'unico elemento influenzato: infatti nel file contenente il linguaggio nativo i parametri del metodo dovranno essere tradotti secondo i tipi della JNI seguendo le regole appena spiegate, inoltre comparirà subito la variabile `env` per consentire il passaggio di dati per riferimento.
I suddetti aspetti sono esemplificati da questo semplice esempio. Il metodo Java

```
float prova(int var)
```

della classe `Tesi` presente al package `it.dei` viene tradotto nel metodo NDK

```
JNIEXPORT jfloat JNICALL Java_it_dei_Tesi_prova(JNIEnv *env,
jint var)
```

`JNIEXPORT` indica che tale metodo deve essere esportato in modo da essere visibile in tutto in pacchetto.

1.6 Aspetti tecnici dell'NDK

Il codice nativo, non essendo Java, non può essere interpretato dalla JVM, quindi viene eseguito all'esterno della stessa dopo che il compilatore ha tradotto le istruzioni C/C++ in linguaggio assembly per architetture ARM. Questo elemento non è assolutamente da trascurare: Java viene eseguito da un compilatore Just-in-Time, mentre il linguaggio della JNI utilizza i compilatori standard C/C++, con parametri differenti e possibilità di ottimizzazione del codice.
In queste condizioni si riscontrano codici molto differenti nella fase di progettazione: in C/C++ l'assenza di un Garbage Collector costringe a liberare manualmente le strutture dati utilizzate e ad allocare strutture pesanti in termini di spazio, come array, matrici e struct, sfruttando l'allocazione dinamica, tramite malloc e calloc, in modo da non generare un *segmentation fault* per aver occupato tutta la memoria dedicata ai dati statici del programma. È pertanto corretto affermare che la programmazione ad un livello di astrazione più basso di quello offerto da Java risulta essere più rigida nella sua struttura.
Ovviamente non vi sono soltanto inconvenienti, infatti l'assenza di un organo di controllo libera l'applicazione dai vincoli di spazio: ogni applicazione Java destinata all'esecuzione Android può avere un heap di al massimo 24MB, spazio non troppo vasto se

si pensa alla possibilità di operare con array multidimensionali di double. I linguaggi nativi, proprio non essendo limitati dalla JVM, non hanno limiti di spazio, se non sui dati statici, ossia è possibile allocare dinamicamente qualunque spazio desiderato, in linea teorica fino al riempimento della memoria interna del dispositivo.
Uscire dalla JVM Dalvik non comporta solo questi aspetti: infatti permette l'utilizzo al 100% delle potenzialità della CPU, liberata dal peso della macchina virtuale che resta comunque un'applicazione per quanto ben sviluppata ed ottimizzata, quindi velocizzando almeno in linea teorica l'esecuzione del programma. Tuttavia una complicazione della struttura del programma, non più compatta e lineare, con tanto di metodi da cercare in file differenti nemmeno presenti nella stessa directory, rende il debug molto più complesso: infatti un errore logico di programmazione nel programma nativo non è monitorabile tramite debug Java al punto tale di non lanciare nemmeno eccezioni segnalando il temine di spazio o per aver tentato di accedere ad un'allocazione di memoria errata, generando il più delle volte un segnale di `segmentation fault (11)` o un errore di memora insufficiente per allocare una struttura di dimensioni eccessive e chiudendo l'applicazione, ma senza altre spiegazioni, al contrario dell'ambiente Java che fornisce addirittura la riga di codice che ha generato l'errore, oltre al tipo dello stesso.

1.7 Le differenze tra C/C++ e Java

Tralasciando il linguaggio Assembly che permette un livello ridotto di astrazione dalla macchina, la differenza prestazionale tra i linguaggi C/C++ e Java si fonda non solo sulla diversa filosofia degli sviluppatori dei linguaggi, ma anche sulla compilazione degli stessi.
La filosofia C è vincolata al tempo nel quale in linguaggio venne sviluppato: l'ancora scarsa diffusione dei calcolatori nel 1972 (quando i computer erano proprietà esclusiva di laboratori e delle scuole più prestigiose) portò il ricercatore Dennis MacAlistair Ritchie a sviluppare un linguaggio più semplice e più "potente" dell'Assembly, volto ad ottenere prestazioni eccellenti, ma senza occuparsi troppo della portabilità del file eseguibile.
Java, introdotto al pubblico soltanto nel 1995, doveva far fronte ad una realtà totalmente differente: i personal computer oramai erano diffusi a livello capillare nella società (nel 1993 era stato introdotto il World Wide Web rivoluzionando il concetto di Internet e ampliando le potenzialità dei computer) ed erano presenti sul mercato diverse architetture hardware. Java viene proposto come un linguaggio più semplice rispetto ai propri predecessori, rinunciando ad alcuni costrutti di programmazione avanzata (primo fra tutti il concetto di puntatore), strutturato per essere estremamete portabile

e con una migliore tipizzazione e di conseguenza un controllo dell'errore più efficiente. Dalle differenti caratteristiche di C e Java derivano strutture di compilazione radicalmente differenti. Java è un linguaggio quasi completamente interpretato e basato sull'esecuzione del codice in una "Macchina Virtuale", ossia un sistema fittizio con caratteristiche indipendenti dalla macchina fisica su cui essa è installata. Il sorgente viene tradotto dal compilatore in *.class*, quindi l'interprete Java lo può convertire in *bytecode*. Quindi il codice bytecode viene compilato dalla JVM installata nell'ambiente Java JRE (*Java Runtime Enviroment*) ed eseguito. Proprio perché tale compilazione, a differenza di un sorgente C, avviene "al momento" dell'esecuzione il compilatore Java della JVM, viene detto *Just-in-Time Compiler*.

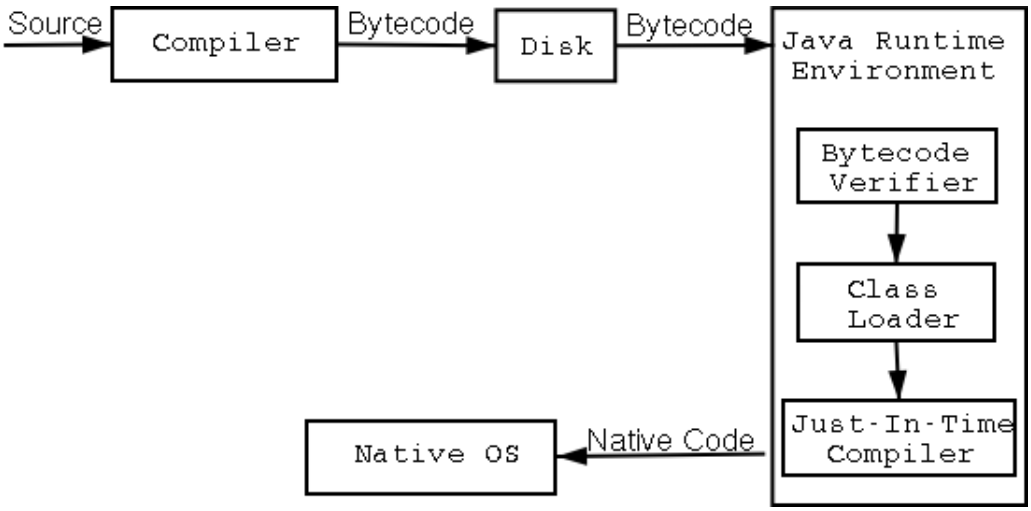


Figura 1.2: Compilazione di un sorgente Java

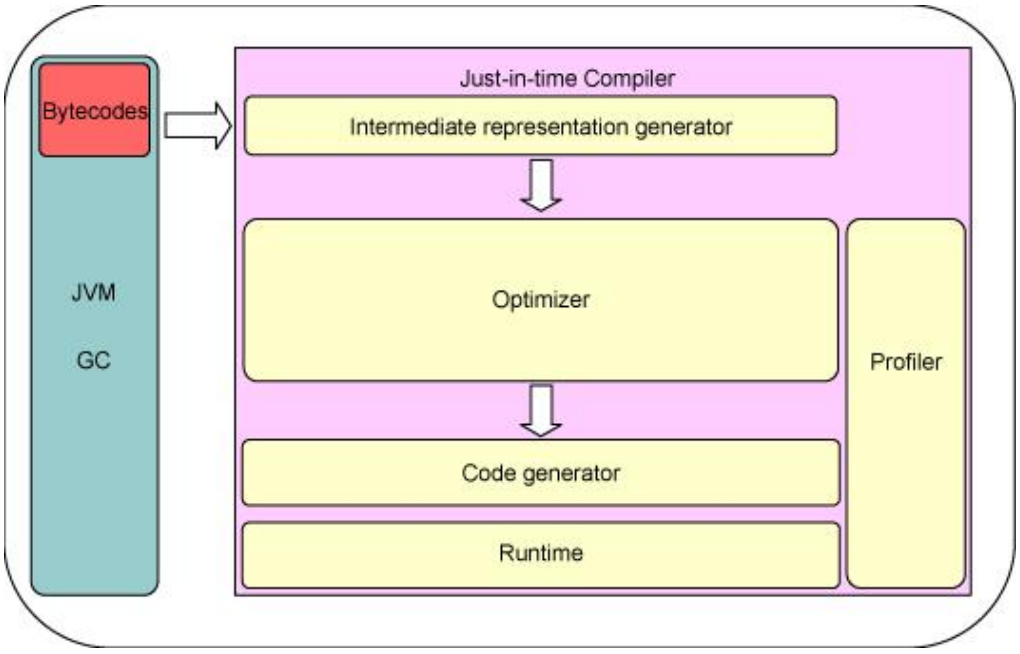


Figura 1.3: Componenti della Java Virtual Machine

Il linguaggio C invece si basa su una compilazione nella quale il sorgente viene tradotto in *assembly*, facendo quindi riferimento all'architettura della macchina sulla quale viene eseguita tale compilazione, oltre che al compilatore utilizzato (ad esempio il diffuso GNU `gcc`). Questa operazione può essere svolta fornendo dei parametri per dirigere l'operazione, tra cui sopprimere o rendere attivi alcuni *warnings*, indicare un livello di ottimizzazione del codice (ossia quanto deve essere rielaborato il codice risultante dalla compilazione) e attivare o meno alcuni messaggi di debug.
Quindi l'assemblatore importa le librerie necessarie al programma, traduce l'assembly in *modulo oggetto*, quindi il linker costruisce il programma eseguibile che può essere eseguito senza macchine virtuali di sorta.
Vista la presenza del linguaggio assembly, risulta logico che un tale programma eseguibile non è indipendente dalla macchina per cui viene eseguita la compilazione al punto tale che il risultato finale è vincolato all'architettura ed al sistema operativo della macchina.
Tale sistema di compilazione è riportato nello schema sottostante.

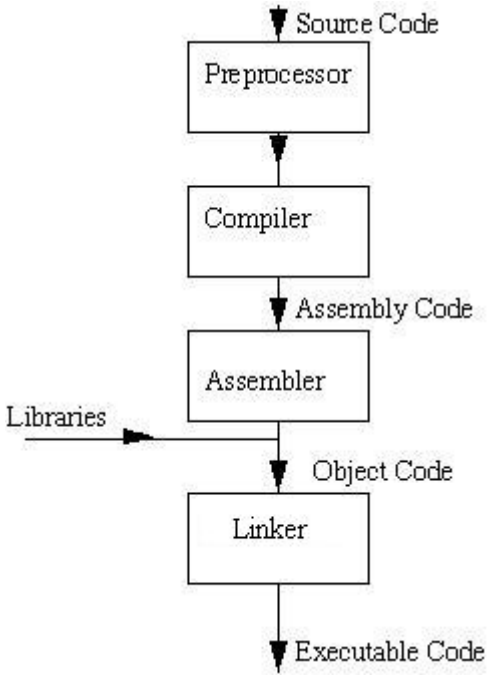


Figura 1.4: Compilazione di un sorgente C

Capitolo 2

Metodologie

2.1 Gli strumenti utilizzati nell'esperienza

Prerequisito fondamentale è l'installazione sia dell'*Android SDK* (spesso abbreviato in ADK), sia dell'*NDK*, reperibili entrambi dal sito ufficiale di Android Developers.
Nonostante l'intera programmazione possa essere condotta tramite strumenti elementari come notepad e comandi nella console a Unix (per l'utilizzo del compilatore GNU di C e C++), la necessità di un debugger e la possibilità di avere un logcat real-time invita all'utilizzo di strumenti più avanzati e completi. In questo caso è stato scelto l'IDE (Integrated Development Environment) *Eclipse*, uno dei più famosi strumenti gratuiti di sviluppo per applicazioni Java, sfruttando anche i plugin scaricabili e facilmente configurabili.
Il primo strumento aggiuntivo è proprio l'ADK che fornirà, oltre agli strumenti di compilazione in file di installazione Android (file .apk), anche la possibilità di utilizzare degli emulatori di sistemi Android dotati di caratteristiche impostabili. Tali emulatori sono applicazioni grafiche che replicano l'esatto funzionamento di un dispositivo, con impostazioni modificabili, quali la quantità di memoria RAM, la versione di Android montata e la dimensione dello schermo.
Gli altri plugin di Eclipse sono tutti orientati all'NDK ed alla programmazione in C/C++. *Sequoyah* è uno strumento molto semplice quanto utile che permette con un semplice click di mouse di aggiungere il supporto NDK alla propria applicazione, invece che creare i collegamenti e i file opportuni a mano. Inoltre permette l'esecuzione degli esempi in linguaggio nativo contenuti nel pacchetto NDK.
L'ultimo plugin, *Eclipse CDT* (C/C++ Development Tooling) è opzionale e mette a disposizione un ambiente piuttosto spoglio ed essenziale per la programmazione in C o C++. Il fatto che sia opzionale deriva dall'installazione già effettuata dell'NDK che contiene già i compilatori necessari (generalmente `gcc`, ossia il compilatore GNU).
In ambiente Windows servirà anche il modo di utilizzare il compilatore gcc, che per definizione necessita dei comandi Unix. Pertanto sarà necessario installare ed aggiungere

alle variabili d'ambiente *Cygwin*, una console ai comandi Unix. Ecplise è già configurato per svolgere le operazioni che servono per la compilazione dei file NDK tramite tale console. [3]
Il dispositivo utilizzato nell'esperienza, in modo da rilevare i dati direttamente e non attraverso un emulatore, è il Motorola Atrix 4G [5], uno smartphone di fascia media, con un processore dual-core Nvidia Tegra 2 da 1GHz e 1GB di RAM, oltre a 10 GB di memoria interna. La piattaforma utilizzata per i test è Android Gingerbread 2.3.4, in versione stock rilasciata da Motorola. Durante i test sul dispositivo è stata attivata la modalità aereo, disattivando quindi le reti cellulari, ed è stato lasciato a "riposo" garantendo la totalità delle risorse (per quanto possibile) del dispositivo ai programmi da testare.

Figura 2.1: Viste del Motorola Atrix, dispositivo utilizzato per le prove

2.2 La struttura dei programmi

Dato che l'utilizzo estensivo dell'NDK viene sconsigliato anche dalla documentazione ufficiale [1], il linguaggio nativo viene utilizzato solo per i metodi sui quali si baserà il confronto, ossia sulle operazioni che richiedono un uso intensivo della CPU.
I programmi sono quasi assimilabili a test da sforzo per lo smartphone: vengono effettuate 10 prove nelle quali viene ripetuto lo stesso algoritmo con varie taglie di dati, quindi ne viene registrato il tempo in millisecondi su un apposito file di testo. Le taglie dei dati sono state portate quasi al massimo consentito dall'heap Java in quanto tale ambiente viene utilizzato in ogni caso per inizializzare le strutture. Da una parte, quindi, vi saranno progetti scritti totalmente in Java, mentre dall'altra progetti che svolgeranno la stessa funzione, ma sfruttando per certe parti il supporto del linguaggio nativo. In particolare, solo i metodi che faranno il prodotto delle matrici e l'ordinamen-

to dell'array saranno delegati all'NDK, in modo da effettuare confronti il più realistici possibile dei metodi che richiedono massicci calcoli da parte della CPU.
I metodi nativi si troveranno a svolgere più operazioni dei corrispettivi in Java, poiché dovranno recuperare le strutture dati, svolgere la propria funzione, quindi liberare le strutture locali. La maggior parte del lavoro iniziale quindi è dovuta alla filosofia dell'esperimento, che costringe il metodo JNI a non avere a priori i dati già disponibili. È pertanto necessaria un'analisi dei volumi delle operazioni che verrà effettuata per il Si vedrà tuttavia che tali operazioni dal punto di vista quantitativo sono totalmente ininfluenti dal punto di vista percentuale, come dimostrato nella tabella 3.1 nella sezione "Prodotto canonico di matrici".

Capitolo 3

Prodotto canonico di matrici

Tra le operazioni che sfruttano massicciamente la CPU certamente il prodotto canonico o righe per colonne tra due matrici è uno degli esempi più calzanti: infatti è un algoritmo di complessità O(n^3), con n taglia del lato di una matrice, a struttura rigida ossia senza casi ottimi o pessimi.

```
A, B matrici [n] * [n]
R matrice [n] * [n] //matrice per i risultati
i, j, k interi

for i = 0 to n
   for j = 0 to n
      for k = 0 to n
         R[i][j] = R[i][j] + A[i][k] * B [k][j]
```

3.1 L'algoritmo

L'algoritmo del programma è molto semplice: vengono allocate due matrici di interi in Java di dimensione ogni volta differente e riempite con valori casuali, poi ne viene effettuato il prodotto canonico. Il programma completamente sviluppato in Java, sfruttando il fatto che le matrici sono variabili di istanza interne alla classe quindi immediatamente raggiungibili, inizia a conteggiare il tempo, effettua il calcolo del prodotto, calcola il tempo utilizzato e salva tale tempo sul file di testo. Da notare che viene conteggiato soltanto il tempo utilizzato per il calcolo effettivo, non il tempo per l'allocazione ed in riempimento delle matrici.
Il programma che si affida al metodo sviluppato in C++ dovrà inviare le matrici come parametri alla funzione scritta in linguaggio nativo, in quanto essa si trova logicamente nello stesso modulo ma all'esterno della classe. Le matrici vengono interpretate come `jobjectArray` ossia come un array di oggetti di provenienza Java. Il primo aspetto

è quindi tradurre in modo corretto le matrici, dunque innanzitutto copiarne i riferimenti alle aree di memoria, prelevarle riga per riga forzando l'interpretazione come `jintArray` (array di interi provenienti da Java) quindi per ogni elemento effettuare una copia direttamente sulle matrici di supporto locali. Queste operazioni, essendo presenti soltanto nell'implementazione del metodo in linguaggio nativo, comportano un carico più elevato di lavoro rispetto allo stesso metodo in Java, portando immediatamente a pensare che un metodo così costituito non possa in alcun caso essere più rapido di uno che compie nettamente meno operazioni.

3.2 Il codice

Per una questione di completezza viene riportato il codice del metodo scritto in C, in modo che saltino subito all'occhio gli elementi stilistici fino ad ora affrontati solo nella teoria

```
JNIEXPORT jdouble JNICALL
Java_it_unipd_tesi_ProdottoMatriciCActivity_prodottoMatrici
(JNIEnv * env, jobject obj, jobjectArray m1, jobjectArray m2, jint dim){

    timespec inizioP, fineP, start, end;
    long t;
    int i,j,k;
    int **m1Locale,**m2Locale, **prodotto; //matrici di copia

    //tempo di inizio del programma
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);

    /*allocazione dinamica delle matrici*/
    m1Locale = (int**) malloc (dim * sizeof(int));
    m2Locale = (int**) malloc (dim * sizeof(int));
    prodotto = (int**) malloc (dim * sizeof(int));
    for(i=0; i<dim; i++){
        m1Locale[i] = (int*) malloc (dim * sizeof(int));
        m2Locale[i] = (int*) malloc (dim * sizeof(int));
        prodotto[i] = (int*) malloc (dim * sizeof(int));
    }//for

    /*Ricostruzione delle matrici*/
    for(i=0; i<dim; i++) {
        jintArray riga1= (jintArray)env->GetObjectArrayElement(m1, i);
        /*popolazione riga della copia di m1*/
        jint *element1=env->GetIntArrayElements(riga1, JNI_FALSE);
        for(j=0; j<dim; j++) {
            m1Locale[i][j]= element1[j];
        }//for

        env->ReleaseIntArrayElements(riga1, element1, JNI_ABORT);
        env->DeleteLocalRef(riga1);

        /*popolazione riga della copia di m2*/
        jintArray riga2= (jintArray)env->GetObjectArrayElement(m2, i);
        jint *element2=env->GetIntArrayElements(riga2, JNI_FALSE);
        for(j=0; j<dim; j++) {
```

```
            m2Locale[i][j]= element2[j];
        }//for

        env->ReleaseIntArrayElements(riga2, element2, JNI_ABORT);
        env->DeleteLocalRef(riga2);
    }//for
    /*esecuzione del prodotto della matrici*/
    for(i=0; i<dim; i++)
        for(j=0; j<dim; j++)
            for(k=0; k<dim; k++)
                prodotto[i][j] += m1Locale[i][k]*m2Locale[k][j];

    /*liberazione dello spazio delle matrici*/
    for(i=0 ; i<dim; i++){
        free(m1Locale[i]);
        free(m2Locale[i]);
        free(prodotto[i]);
    }//for
    free(m1Locale);
    free(m2Locale);
    free(prodotto);

    //tempo di fine del programma
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);

     //restituzione differenza in millisecondi
    return (jdouble)diff(start,end);
}//Java_it_unipd_ProdottoMatriciCActivity_prodottoMatrici
```

3.3 I volumi di operazioni

Il numero delle operazioni per la funzione in C++ sono elencati dalla tabella 3.1 che riporta una stima del numero per il calcolo, quelle per la copia e la cancellazione delle matrici locali ed il rapporto tra i due valori. I dati non sono ovviamente accurati alla signola operazione ma sono state utilizzate le complessità computazionali, valutando infatti che la copia di una matrice richiede una complessità lineare ($O(n^2)$), con n taglia del lato delle matrici ed è un'operazione da effettuare per entrambe le matrici passate tramite riferimento, mentre il prodotto canonico richiede per definizione una complessità $O(n^3)$ infatti per ogni elemento della matrice risultato (quindi n^2) devo scandire con lo stesso passo la riga della prima matrice e la colonna della seconda (per cui n operazioni).

Si intuisce che il fatto di dover effettuare una copia non influisce in modo rilevante sulle prestazioni del programma, aggiungendo infatti una percentuale piuttosto ridotta alle operazioni. Quindi, se il C++ fosse davvero in grado di eseguire le istruzioni per il calcolo in minor tempo rispetto a Java, sarebbe preferibile, nonostante un numero di operazioni comunque superiori (Java non deve ricopiare matrici, né liberarle una volta svolti i calcoli).

Lato	N elementi	Copia matrici	Prodotto	Rapporto
100	10000	20000	1000000	2,00000%
150	22500	45000	3375000	1,33333%
200	40000	80000	8000000	1,00000%
250	62500	125000	15625000	0,80000%
300	90000	180000	27000000	0,66667%
350	122500	245000	42875000	0,57143%
400	160000	320000	64000000	0,50000%
450	202500	405000	91125000	0,44444%
500	250000	500000	125000000	0,40000%
550	302500	605000	166375000	0,36364%
600	360000	720000	216000000	0,33333%
650	422500	845000	274625000	0,30769%
700	490000	980000	343000000	0,28571%
750	562500	1125000	421875000	0,26667%
800	640000	1280000	512000000	0,25000%
850	722500	1445000	614125000	0,23529%
900	810000	1620000	729000000	0,22222%
950	902500	1805000	857375000	0,21053%

Tabella 3.1: Rapporti percentuali tra le operazioni di recupero delle matrici e le operazioni necessarie per il calcolo, in rapporto alla taglia delle stesse

3.4 I grafici temporali

Invece che riportare tabelle interminabili di valori di tempi in millisecondi, si è preferito riportare un risultato grafico, certamente più immediato e rappresentativo di semplici valori numerici.
Il seguente grafico (3.1) riporta per ogni taglia i valori temporali a confronto. I valori rappresentati sono valori di media, quindi risultati piuttosto realistici e non derivanti da un semplice "caso" o da uno stato particolare del dispositivo. Si vedrà in seguito un'analisi più accurata di parte dei dati numerici. Il grafico evidenzia un netto divario di prestazioni tra le ottimizzazioni varie di C rispetto al programma Java. Le serie di dati relative al linguaggio C sono quattro perché ve ne è una per ogni possibile grado di ottimizzazione del codice, da quella con totale assenza di ottimizzazione fino a quella di grado 3. Tale aspetto verrà esplicato meglio nella sezione "Ottimizzazioni".

3.5 Analisi dei risultati

Il grafico 3.1 mostra certi andamenti, ma nonostante sia ricavato dalla media temporale di una decina di prove, questo non indica per forza un grafico accurato e soprattutto corretto nel suo insieme. Infatti se per ogni prova vi fosse un'elevatissima dispersione l'esperimento sarebbe un fallimento, in quanto non rileverebbe misure attendibili. Pertanto, sfruttando per comodità le funzioni di un foglio di calcolo elettronico è stata

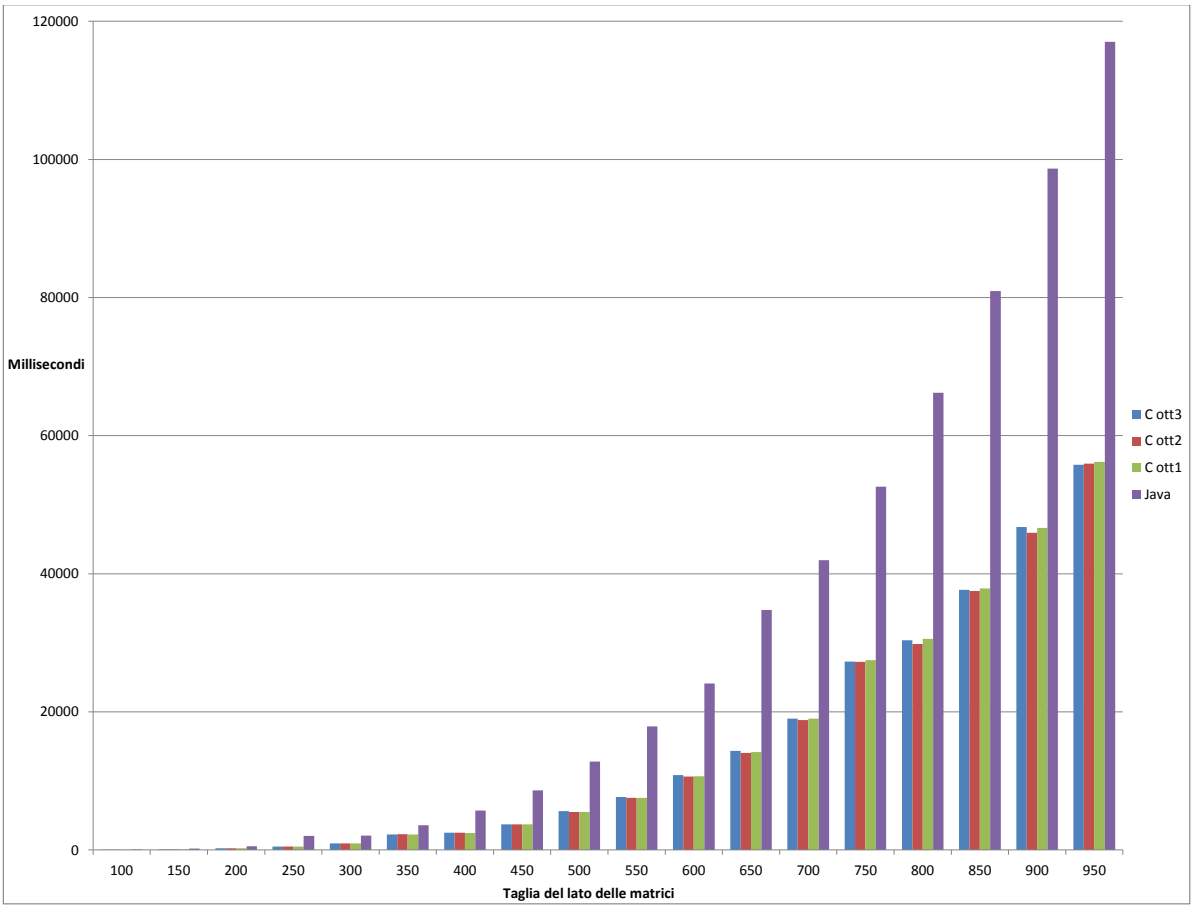


Figura 3.1: Tempi del prodotto canonico per taglia

ricavata la varianza σ^2 e lo scarto quadradico medio σ.
L'attendibilità è quindi riportata a metodi statistici: la ripetizione indipendente di un esperimento di misura di tempi viene genericamente approssimata con una distribuzione di probabilità di Gauss, anche nota come "Normale". Questa considerazione è derivata dal *Teorema centrale del Limite*: una successione di variabili aleatorie reali indipendenti (e risultanti dallo stesso esperimento) a media e varianza finite tendono ad una distribuzione Normale. Tra le caratteristiche di una distribuzione normale ve ne sono alcune fondamentali:

- Dominio e codominio nei reali
- Simmetria rispetto alla media μ
- $P[\mu - 3\sigma < x < \mu + 3\sigma] \approx 99.7\%$
- $P[\mu - 2\sigma < x < \mu + 2\sigma] \approx 95.5\%$
- $P[\mu - 1\sigma < x < \mu + 1\sigma] \approx 68.3\%$

Sfruttando le ultime tre proprietà potremo quindi apprezzare quanto le misure siano accurate: trovando un valore distante dal tempo medio di più di 3σ, tale misura sarà

da scartare.
La tabella sottostante riporta parte dell'analisi svolta per i dati ottenuti per l'ottimizzazione di grado 0 del C (ossia con l'assenza di ottimizzazione). Come si può vedere,

	100	200	300	400	500	600	700	800	900
prova0	17,66	248,56	993,46	2699,37	5613,52	10580,63	19105,57	30324,36	45210,70
prova1	17,96	241,41	939,40	2792,93	5681,43	10474,08	19106,10	31288,35	44958,30
prova2	17,88	241,70	940,20	2655,69	5686,92	10600,69	19260,63	31018,70	46871,56
prova3	18,24	241,56	941,82	2799,12	5677,71	10699,88	18430,14	31214,13	45796,12
prova4	18,03	243,04	940,86	2657,96	5630,70	10615,81	19468,49	30934,44	45547,34
prova5	17,62	233,33	912,23	2660,74	5447,05	10642,73	19205,01	29889,59	47866,05
prova6	17,60	234,47	912,02	2713,49	5687,13	10017,41	18976,40	30783,80	47667,63
prova7	17,99	241,50	941,84	2794,38	5683,18	10644,74	19154,44	31970,08	46935,70
prova8	17,41	234,33	973,08	2794,18	5658,46	10418,57	18416,48	29355,47	45623,54
prova9	17,68	234,13	941,54	2798,00	5620,89	10816,27	19278,33	29900,96	46221,92
MEDIE	17,81	239,40	943,64	2736,59	5638,70	10551,08	19040,16	30667,99	46269,89
VARIANZE	0,06	23,02	538,39	3791,22	4813,32	42545,67	110282,76	562095,17	931545,80
SQM	0,24	4,80	23,20	61,57	69,38	206,27	332,09	749,73	965,17
MIN AMM	**17,09**	**225,01**	**874,03**	**2551,87**	**5430,57**	**9932,28**	**18043,89**	**28418,80**	**43374,39**
MAX AMM	**18,52**	**253,80**	**1013,25**	**2921,30**	**5846,83**	**11169,88**	**20036,42**	**32917,18**	**49165,38**

Figura 3.2: Tempi, media, varianza, scarto quadratico medio e massimi e minimi consentiti per l'Ottimizzazione di grado 0 del C

non vi sono valori che verranno scartati, inoltre gli scarti non sono molto elevati (non più del 3% rispetto alle medie relative) quindi evidenziando curve molto strette, sintomo di una distribuzione molto raccolta intorno alla media. Pertanto, i dati ricavati dagli esperimenti risultato accettabili.

3.6 Le ottimizzazioni

Il grafico riporta le varie ottimizzazioni del C, tuttavia non mostra come utili i gradi di ottimizzazione, infatti i tempi del grado 1 sono estremamente simili, se non addirittura inferiori, ai tempi del grado 3 (quello massimo). Questo è dovuto al fatto che il codice C

per tale metodo è estremamente semplice e breve, per cui il compilatore non è in grado di ottimizzarlo ulteriomente. In ambiti ben differenti tuttavia tali ottimizzazioni non sono elementi da trascurare: se si utilizzano costrutti complessi (ad esempio `struct` e operatori ternari), essi portano a scrivere codice elegante, ma certamente non efficiente, che quindi può essere velocizzato dai diversi gradi di ottimizzazione. Non va inoltre tralasciata la lunghezza del codice: tale aspetto può risultare drammatico in programmi di una certa complessità, che fanno riferimento a diverse librerie, quindi il compilatore per ottenere risultati migliori nell'interpretazione del listato C in istruzioni assembly può cambiare l'ordine di certe operazioni o dividere operazioni complesse in alcune più semplici e più veloci da eseguire (le divisioni ad esempio possono essere ridotte a shift logici e moltiplicazioni, i cicli `for` ridotti a *salti*,...).

3.7 Grafici temporali con matrici di double

Per un confronto più completo è stato ripetuto l'esperimento con matrici di *double*, dati certamente più complessi di semplici interi, come la figura sottostante (3.3) può confermare: infatti, mentre gli interi hanno un bit per il segno ed i restanti utilizzati per esprimere in complemento a 2 il numero, la struttura dei floating point a doppia precisione, oltre al *bit di segno*, riporta un *esponente* a cui sottrarre 1023 ed elevare il numero ottenuto aggiungendo 1 alla *mantissa* intesa come parte decimale del numero [4]. In pratica il numero si ottiene seguendo la formula in figura 3.4.

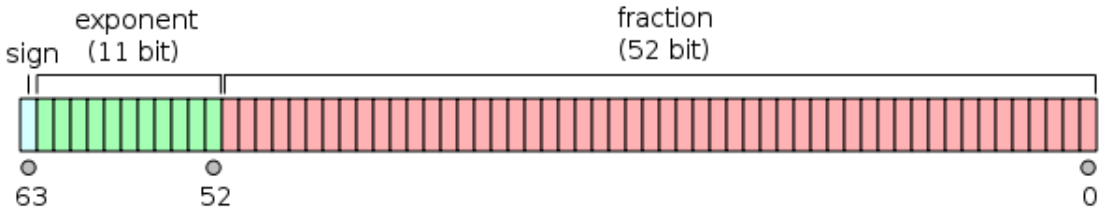


Figura 3.3: Struttura di una variabile double

$$\text{value} = (-1)^{\text{sign}}(1 + \sum_{i=1}^{52} b_{-i}2^{-i}) \times 2^{(e-1023)}$$

Figura 3.4: Calcolo di un numero partendo da una variabile double

Per questo esperimento sono state ovviamente adottate le stesse modalità per garantire risultati accettabili e non "frutto di un caso": prove ripetute ed analisi statistica dei dati.
Il grafico ottenuto (figura 3.5) evidenzia quanto stavolta il tempi di Java siano vicini a C, evidenziando tuttavia un ritardo crescente rispetto alla taglia delle matrici. È quindi corretto affermare che anche in questo caso l'utilizzo di un programma strutturato su elementi NDK è vantaggioso rispetto ad un programma in puro Java.

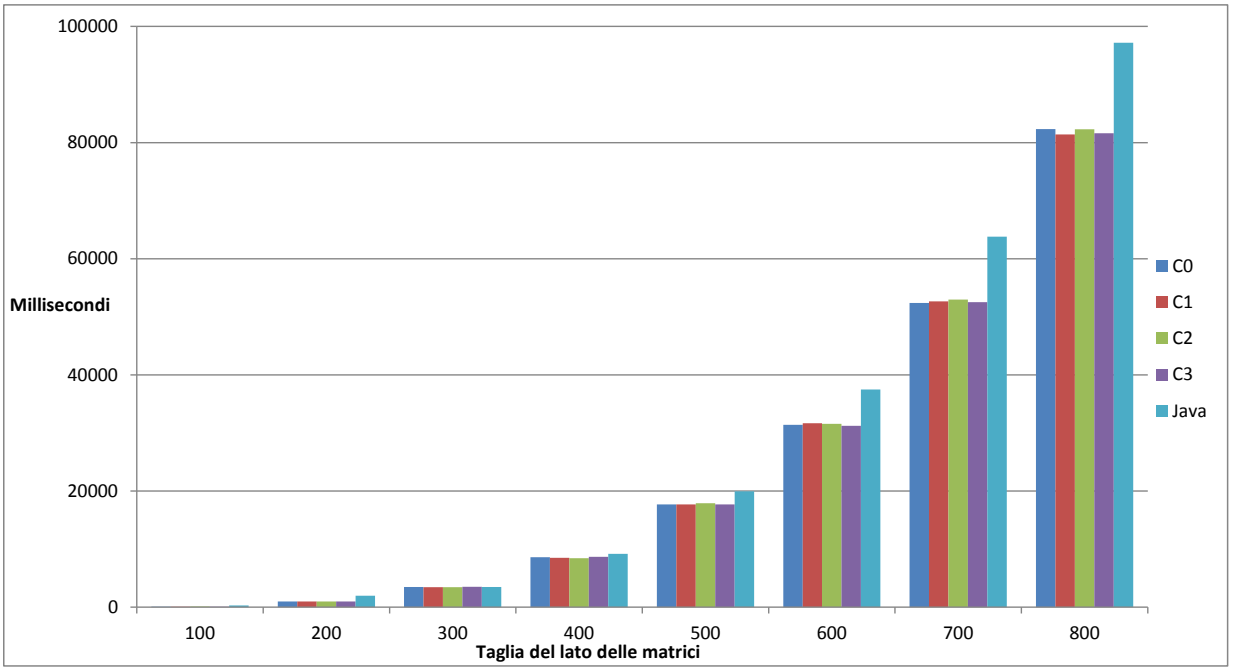


Figura 3.5: Grafico temporale di Java e delle ottimizziazioni di C in rapporto alla taglia del lato delle matrici di double

Capitolo 4

Quicksort

Tale algoritmo sviluppato per ordinare una sequenza di valori (un array di *double* in questo caso) si basa sul concetto di "divide et impera", ossia dividere il problema in sotto-problemi facilmente risolvibili e quindi unire le soluzioni. Si basa sul concetto di scegliere un elemento detto "pivot", spostare gli elementi minori di esso a sinistra e quelli maggiori a destra, senza altro particolare ordine, quindi ripetere per le sotto-sequenze di destra e di sinistra fino al loro esaurimento.
L'algoritmo in pseudo-codice è il seguente

```
begin
    sia A vettore
    sia aj elemento di indice j
    sia ai elemento di indice i
    if n = 1 then return A
    else
        scegli il pivot ak
        costruisci il vettore A1 con ai tali che ai <= ak
        costruisci il vettore A2 con aj tali che aj > ak
        A1 = Quicksort(A1)
        A2 = Quicksort(A2)
end
```

4.1 Il codice

Nonostante esistano librerie Java che implementano l'algoritmo quicksort, è stato ritenuto opportuno implementare l'algoritmo manualmente per avere conoscenza totale del codice utilizzato.
Come già fatto per il prodotto di matrici, viene riportato il sorgente del metodo scritto in C. Tale funzione tuttavia si appoggia notevolmente ad altre funzioni definite nel file JNI.

```
JNIEXPORT jdouble JNICALL
Java_it_unipd_dei_tesi_QuickSortCActivity_sort
(JNIEnv * env, jobject obj, jdoubleArray array, jint dim) {
     timespec start, end;
     int i; //indici
     jdouble *a; //array di copia

    //inizio del programma
     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);

     a = env->GetDoubleArrayElements(array, JNI_FALSE);
     int d = dim;
     quickSort(a, 0, d-1);
     env->ReleaseDoubleArrayElements(array, a, JNI_ABORT);

    //fine del programma
     clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);
     return diff(start, end);
 }//sort
```

Non sfuggirà che questo listato è estremamente breve a causa delle funzioni richiamate, che tuttavia sono estremamente simili a quelle Java, pertanto non interessanti dal punto di vista dell'argomento di questa trattazione.

4.2 Grafici temporali con array di double

Il seguente grafico riporta i dati temporali a confronto rispettando le stesse modalità presentate nel capitolo 3. Contrariamente a quanto visto per il prodotto di matrici, la differenza tra C e Java è minima, nonostante la funzione in C sia comunque più rapida. Tale aspetto è legato all'implementazione dell'algoritmo: esso infatti è ricorsivo, quindi non si affida solo alla potenza della CPU ma anche all'heap della memoria centrale. Inoltre il quicksort per ordinare n elementi impiega un tempo $O(n)$ per la scansione degli elementi, la fase di divisione è di complessità $O(\log_2(n))$, con una complessità totale di $O(n \log(n))$, mentre il prodotto righe per colonne delle matrici ha complessità $O(n^(3/2))$ con n numero di elementi di ogni matrice quadrata.
Per cui il quicksort, o meglio, l'implementazione sviluppata, non soddisfa le condizioni per richiedere un utilizzo massiccio di uno dei due processori (CPU o GPU, a seconda che sia un'applicazione di calcolo come in questo caso o un'applicazione grafica); ciò

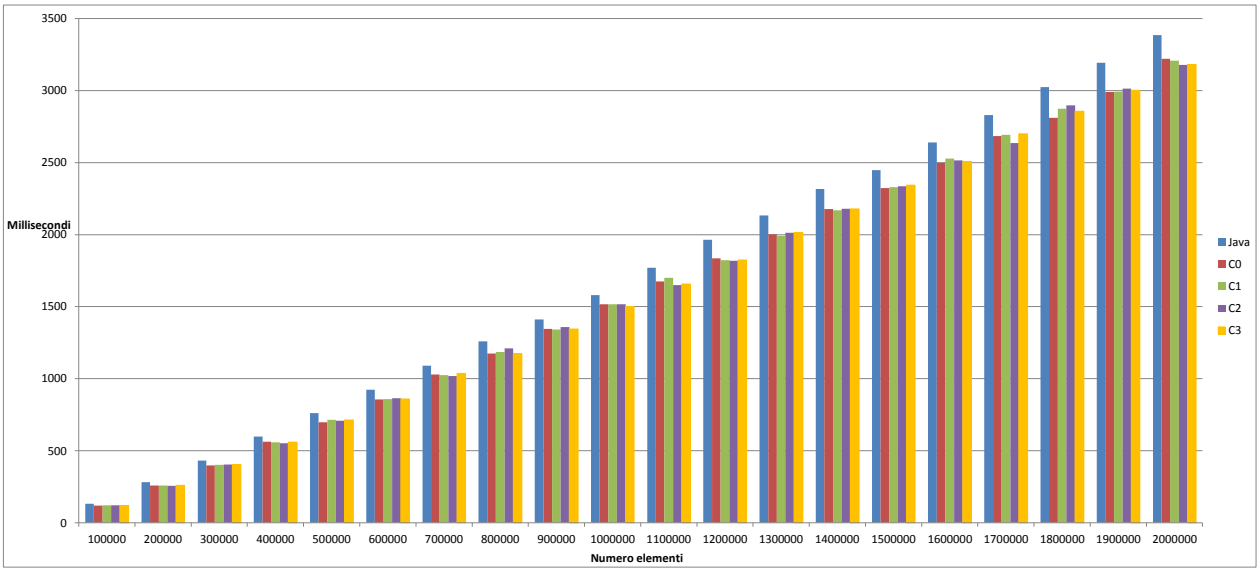


Figura 4.1: Tempi in millisecondi del QuickSort in rapporto al numero di elementi dell'array

nonostante, il programma con il metodo in C si rivela comunque più veloce del suo corrispettivo in puro Java.

4.3 Grafici temporali con array di interi

L'esperimento è stato ripetuto per array di interi anziché di double. Il grafico 4.2 riporta i tempi rilevati con le medesime modalità già descritte.
Stavolta Java è in netto vantaggio rispetto a C; infatti ogni tempo di Java è poco superiore alla metà del tempo in C. Anche con taglie più alte (grafico 4.3) il risultato è il medesimo.
Evidentemente tale algoritmo con array interi implementato in Java risulta troppo semplice per consentire alla JNI di avere un effetto benefico. Come già spiegato questo algoritmo non è ideale per le condizioni che devono rispettare i programmi che utilizzano il supporto NDK e l'utilizzo di variabili più semplici (come di fatto è una variabile *intera* rispetto ad una *double*, ossia a virgola mobile a doppia precisione) riduce ancora di più il carico della CPU, evidenziando quindi i limiti prestazionali dell'utilizzo degli strumenti JNI in casi non ideali.

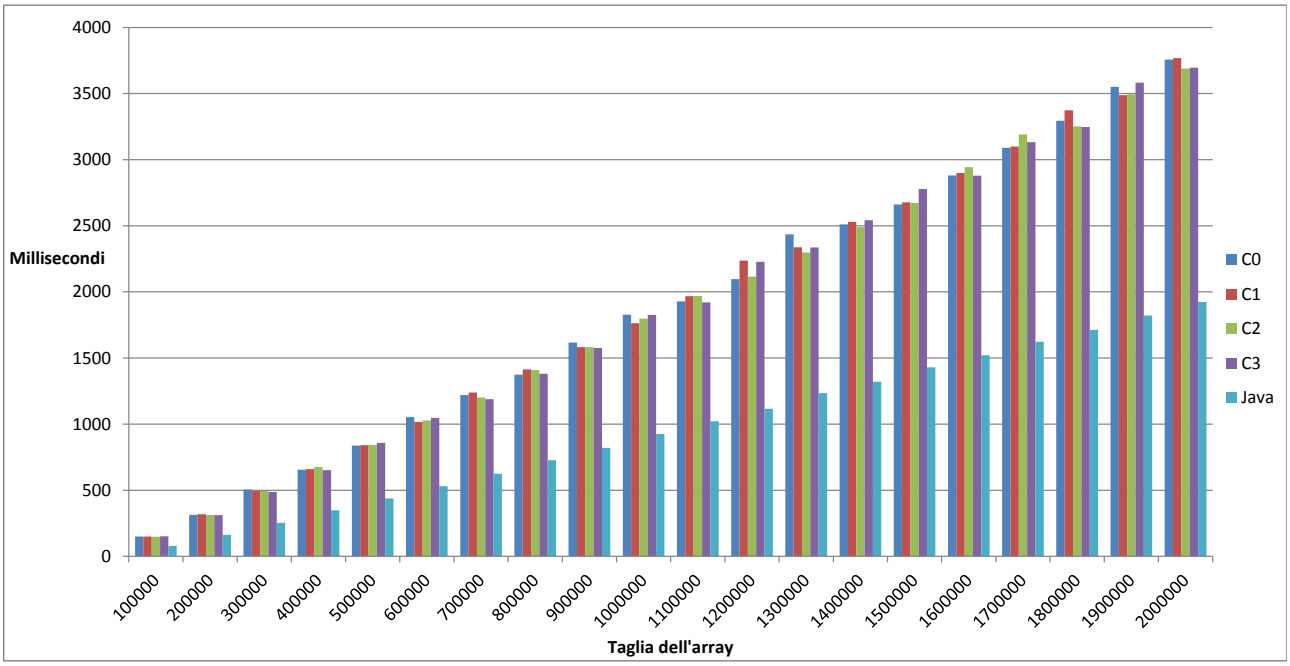


Figura 4.2: Tempi del QuickSort con array di interi per taglia

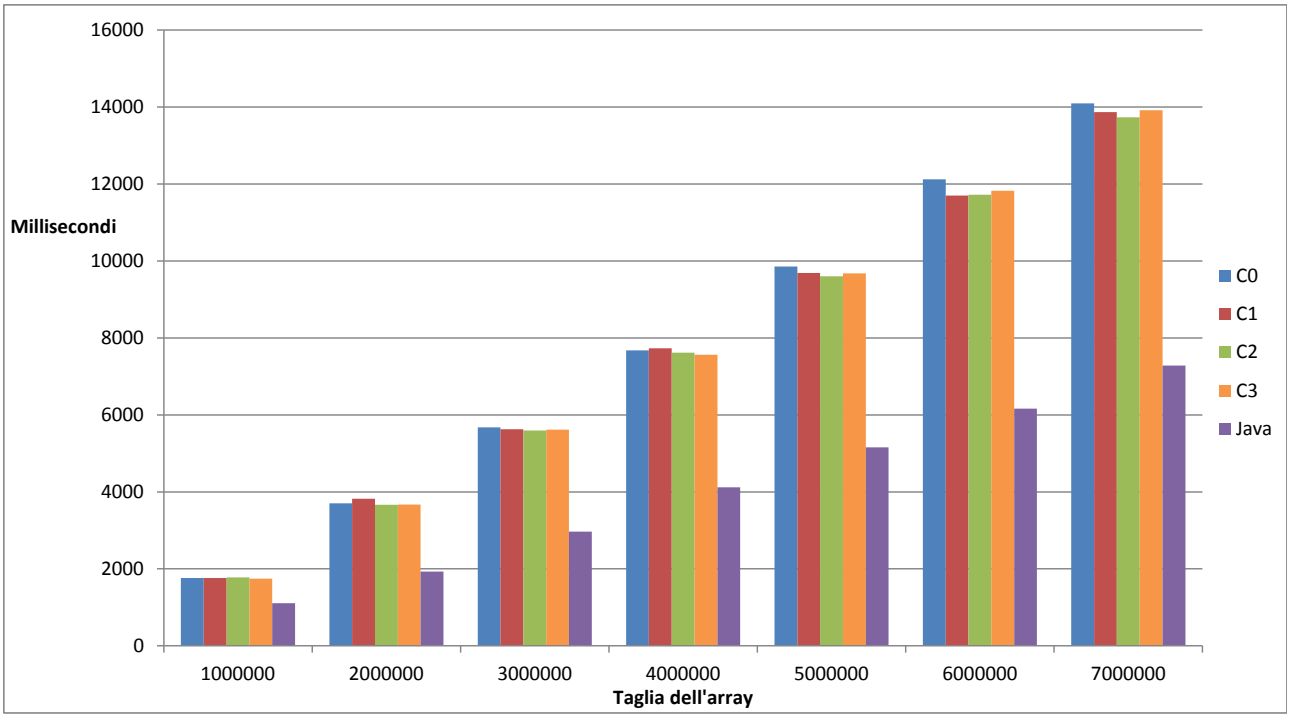


Figura 4.3: Tempi del QuickSort con array di interi per taglia

Capitolo 5

Conclusioni

Le potenzialità e l'effettiva utilità dell'NDK dimostrate negli esperimenti rendono tali strumenti estremamente validi al punto di poter essere consigliati per qualunque applicazione che utilizzi rilevanti calcoli ciclici, complicando di certo lo sviluppo dell'applicazione, ma fornendo un vantaggio in termini di velocità che non può essere trascurato.
Come riportato nella documentazione ufficiale [1], prima di utilizzare l'NDK bisogna soppesare "vantaggi e svantaggi, e questi ultimi sono numerosi"; tuttavia essi sono raggirabili con pochi accorgimenti, infatti con gli strumenti sopra presentati, dopo aver *configurato l'ambiente*, operazione semplice e comunque da eseguire soltanto una volta, resta soltanto la *programmazione dei metodi in linguaggio nativo*, il *collegamento degli stessi* tramite il caricamento della libreria in Java e le eventuali *modifiche al makefile*. Premesso quindi che la programmazione in linguaggio C/C++ richiede un livello di attenzione maggiore e una difficoltà superiore nel debugging del codice, una volta implementato correttamente la funzione, se soddisfa anche parzialmente i requisiti, si potranno riscontrare notevoli miglioramenti prestazionali al confronto del metodo corrispettivo in Java.
Pertanto l'NDK non consiste in soltanto uno strumento aggiuntivo utile soltanto per test temporali teorici oppure che può rivelarsi utile in condizioni estremamente restrittive, bensì risulta *fondamentale ed irrinunciabile in applicazioniche elaborano grandi moli di dati*, come applicazioni con una parte grafica rilevante costruita con OpenGL, libreria grafica interamente sviluppata in C (con l'aggiunta di strutture ad-hoc), che basa la sua natura sul calcolo e sulla manipolazione di matrici di vertici, matrici che devono essere gestite in modo efficiente per evitare che la grafica presenti "lag" dovuti al collo di bottiglia di Java.
Ovviamente non è possibile che il linguaggio nativo rimpiazzi totalmente Java nelle applicazioni Android, in quanto per accedere ai framework principali nei quali si trovano alcune delle classi fondamentali, come la classe Activity che in pratica è l'unica a mettere a disposizione i metodi che garantiscono l'esistenza di un programma su piat-

taforma Android, bisogna utilizzare le strutture previste in Java; inoltre è impossibile negare la semplicità sia di programmazione che di debug del codice Java.
Ancora una volta, quindi, ci si riconduce alla scelta tra semplicità di programmazione e prestazioni. In un ambito tuttavia orientato al minor consumo energetico, programmi che impiegano meno tempo per essere svolti sono senza dubbio preferibili, inoltre in assenza di meccanismi avanzati di risparmio energetico (ad esclusione della riduzione di intensità luminosa dello schermo) è essenziale utilizzare ogni possibile risorsa che altrimenti sarebbe sprecata: non è possibile infatti utilizzare solo una parte del processore o una GPU meno potente (essendovene solo una presente nel dispositivo), pertanto si deve cercare di utilizzare le risorse disponibili nel modo più efficiente possibile, evitando che energia della batteria vada persa per nulla. Viste le considerazioni anche solo di tipo qualitativo ed intuitivo l'NDK rappresenta un modo per superare le inefficienze concettuali del Java, "tornando indietro", ossia a linguaggi di livello più basso (addirittura con la possibilità di utilizzare l'Assembly, linguaggio molto vicino al linguaggio macchina), ma non rinunciando totalmente alla struttura tipizzata e semplice di un linguaggio comunque di alto livello, con meno costrutti complessi e certamente più semplice da programmare per la sua immediata applicazione a partire dagli algoritmi in pseudo-codice.

Bibliografia

[1] Android Developer
http://developer.android.com/tools/sdk/ndk/index.html

[2] Documentazione Oracle di JNI
http://docs.oracle.com/javase/1.4.2/docs/guide/jni/spec/functions.html

[3] Sito web per l'installazione dell'NDK e degli strumenti relativi
http://permadi.com/blog/2011/09/creating-your-first-android-jnindk-project-in-eclipse-with-sequoyah/

[4] Pagina delle variabili double di Wikipedia
http://en.wikipedia.org/wiki/Double-precision_floating-point_format

[5] Pagina ufficiale del Motorola Atrix
http://www.motorola.com/us/consumers/Motorola-ATRIX-4G/72112,en_US,pd.html

Appendice A

In questa sezione sono riportati i sorgenti dei programmi che riguardano il Prodotto Canonico di Matrici, che per brevità, scorrevolezza del testo e utilità sono state omesse nella trattazione.

Sorgente Java

Classe Tesi

Tale classe è la classe iniziale, quella che estende Activity, che si occupa di impostare la dimensione delle matrici, quindi di invocare le funzioni che eseguiranno il calcolo. Inoltre gestisce i tempi che verranno scritti nel file di testo utilizzato come *store* dei tempi.

```
import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;

public class Tesi extends Activity {

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }//onCreate

    public void onResume(){
        super.onResume();
        try{
            //apertura file
```

```
            File file = new File(Environment.
               getExternalStorageDirectory() + File.separator
                + "test.txt");
            file.createNewFile();
            FileOutputStream fos = new FileOutputStream(file)
               ;
            PrintStream out = new PrintStream(fos);
            //10 prove per qualunque valore
            for(int i=0; i<10; i++){
                int dim = 100;
                out.print("prova"+i+" ");
                //un test completo
                do{
                    out.print(" "+(new MatriciInt(dim)).
                       prodottoMatriciale());
                    Log.d("java", "prova "+i+", dim="+dim);
                    dim += 50;
                }while(dim <= 1000);
                out.println("");//new line per la nuova prova
            }//for
            //chiusura stream del file
            out.flush();
            out.close();
            fos.flush();
            fos.close();
        }catch(Exception e){
            Log.e("Tesi","Errore nella gestione del file");
        }//catch
    }//onResume
}//Tesi
```

Classe MatriciInt

Questa classe è quella che si occupa della gestione delle matrici dall'allocazione, al riempimento, fino al prodotto vero e proprio.

```
public class MatriciInt {

    int m1[][], m2[][], dim;

    public MatriciInt(int dim){
        this.dim = dim;
        m1 = new int[dim][dim];
        m2 = new int[dim][dim];
        riempi();
    }//MatriciInt
```

```
    /*metodo per riempire le due matrici*/
    private void riempi(){
        for(int i=0; i<dim; i++)
            for(int j=0; j<dim; j++){
                m1[i][j] = (int) (100* Math.random()); //
                   valore random da 0 a 100
                m2[i][j] = (int) (100* Math.random());
            }//for
    }//riempi

    /*metodo che si occupa di effettuare il prodotto
       matriciale*/
    public long prodottoMatriciale(){
        long inizio = System.currentTimeMillis();
        int ris[][] = new int[dim][dim];
        for(int i=0; i<dim; i++)
            for(int j=0; j<dim; j++)
                for(int k=0; k<dim; k++)
                    ris[i][j] += m1[i][k]*m2[k][j];
        long fine = System.currentTimeMillis();
        return fine - inizio;
    }//prodottoMatriciale
}//MatriciInt
```

Sorgente ibrido

Classe ProdottoMatriciCActivity

Tale classe risulta l'analogo della classe Tesi nel progetto interamente sviluppato in Java, anche se si occupa anche dell'allocazione e del riempimento delle matrici che verranno passate al modulo JNI.

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;

import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;

public class ProdottoMatriciCActivity extends Activity {

    int dim, m1[][], m2[][];

    /** Called when the activity is first created. */
```

```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
}//onCreate

public void onStart(){
    super.onStart();
    try{
        //apertura file
        File file = new File(Environment.
           getExternalStorageDirectory() + File.separator
            + "nativeO3.txt");
        file.createNewFile();
        FileOutputStream fos = new FileOutputStream(file)
           ;
        PrintStream out = new PrintStream(fos);
        //10 prove per qualunque valore
        for(int i=0; i<10; i++){
            dim = 100;
            out.print("prova"+i+" ");
            //un test completo
            do{
                alloca();
                riempi();
                out.print(" "+prodottoMatrici(m1,m2,dim))
                   ;
                Log.d("native", "prova "+i+", dim="+dim);
                dim += 50;
            }while(dim < 1000);
            out.println("");//new line per la nuova prova
        }//for

        //chiusura stream del file
        out.flush();
        out.close();
        fos.flush();
        fos.close();

        return;
    }catch(Exception e){
        Log.e("Tesi","Errore nella gestione del file");
    }//catch
}//onResume

private void alloca(){
    m1 = new int[dim][dim];
```

```
        m2 = new int[dim][dim];
    }//alloca

    private void riempi(){
        for(int i=0; i<dim; i++)
            for(int j=0; j<dim; j++){
                m1[i][j] = (int) (100* Math.random()); //
                    valore random da 0 a 100
                m2[i][j] = (int) (100* Math.random());
            }//for
    }//riempi

    //prototipo dei metodi nativi
    public native double prodottoMatrici(int m1[][], int m2
        [][], int dim);

    //caricamento della libreria (consultare il file .mk per
        il nome)
    static {
        System.loadLibrary("ProdottoMatriciC");
    }
}//ProdottoMatriciCActivity
```

Modulo ProdottoMatriciC

Modulo che contiene la funzione che effettua il prodotto matriciale e quello per calcolare la differenza tra due timestamp, metodo scelto per calcolare il tempo impiegato.

```
#include <string.h>
#include <jni.h>
#include <time.h>

extern "C" {
    JNIEXPORT jdouble JNICALL
        Java_it_unipd_tesi_ProdottoMatriciCActivity_prodottoMatrici
        (JNIEnv * env, jobject obj, jobjectArray m1,
        jobjectArray m2, jint dim);
};

double diff (timespec start, timespec end);

JNIEXPORT jdouble JNICALL
    Java_it_unipd_tesi_ProdottoMatriciCActivity_prodottoMatrici
    (JNIEnv * env, jobject obj, jobjectArray m1, jobjectArray
    m2, jint dim){

    timespec inizioP, fineP, start, end;
```

```
long t;
int i,j,k;
int **m1Locale,**m2Locale, **prodotto; //matrici di copia

clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);//tempo
   di inizio del programma

/*allocazione dinamica delle matrici*/
m1Locale = (int**) malloc (dim * sizeof(int));
m2Locale = (int**) malloc (dim * sizeof(int));
prodotto = (int**) malloc (dim * sizeof(int));
for(i=0; i<dim; i++){
     m1Locale[i] = (int*) malloc (dim * sizeof(int));
     m2Locale[i] = (int*) malloc (dim * sizeof(int));
     prodotto[i] = (int*) malloc (dim * sizeof(int));
}//for

/*Ricostruzione delle matrici*/
for(i=0; i<dim; i++) {
     jintArray riga1= (jintArray)env->
        GetObjectArrayElement(m1, i);
     /*popolazione riga della copia di m1*/
     jint *element1=env->GetIntArrayElements(riga1,
        JNI_FALSE);
     for(j=0; j<dim; j++) {
         m1Locale[i][j]= element1[j];
     }//for

     env->ReleaseIntArrayElements(riga1, element1,
        JNI_ABORT);
     env->DeleteLocalRef(riga1);

     /*popolazione riga della copia di m2*/
     jintArray riga2= (jintArray)env->
        GetObjectArrayElement(m2, i);
     jint *element2=env->GetIntArrayElements(riga2,
        JNI_FALSE);
     for(j=0; j<dim; j++) {
         m2Locale[i][j]= element2[j];
     }//for

     env->ReleaseIntArrayElements(riga2, element2,
        JNI_ABORT);
     env->DeleteLocalRef(riga2);
}//for
/*esecuzione del prodotto della matrici*/
for(i=0; i<dim; i++)
```

```
            for(j=0; j<dim; j++)
                for(k=0; k<dim; k++)
                    prodotto[i][j] += m1Locale[i][k]*m2Locale[k][
                        j];

        /*liberazione dello spazio delle matrici*/
        for(i=0 ; i<dim; i++){
            free(m1Locale[i]);
            free(m2Locale[i]);
            free(prodotto[i]);
        }//for
        free(m1Locale);
        free(m2Locale);
        free(prodotto);

        clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);//tempo di
            fine del programma

        return (jdouble)diff(start,end); //restituzione della
            differenza in millisecondi
    }//Java_it_unipd_ProdottoMatriciCActivity_prodottoMatrici


    /*Funzione per il calcolo della differenza tra due timespec*/
    double diff (timespec start, timespec end){
        timespec tempo;
        if ((end.tv_nsec-start.tv_nsec) < 0){ //correzione per un
            giro di orologio
            tempo.tv_sec = end.tv_sec-start.tv_sec-1;
            tempo.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
        }else{
            tempo.tv_sec = end.tv_sec-start.tv_sec;
            tempo.tv_nsec = end.tv_nsec-start.tv_nsec;
        }//else
        return (double)((double)tempo.tv_sec*1000 + (double)tempo
            .tv_nsec/1000000);
    }//diff
```

Appendice B

In questa sezione sono riportati i sorgenti dei programmi che riguardano il Quicksort di un array.

Sorgente Java

Classe QuickSortJavaActivity

Tale classe estende Activity e gestisce i tempi da risportare nel file di testo, oltre ad invocare i metodi per effetture il programma.

```
import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;
import java.io.File;
import java.io.FileOutputStream;
import java.io.PrintStream;

public class QuickSortJavaActivity extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }//onCreate

    public void onResume(){
        super.onResume();

        try{
            //apertura file
            File file = new File(Environment.
               getExternalStorageDirectory() + File.separator
                + "quickJava.txt");
            file.createNewFile();
            FileOutputStream fos = new FileOutputStream(file)
               ;
```

```
            PrintStream out = new PrintStream(fos);
            //10 prove per qualunque valore
            for(int i=0; i<10; i++){
                int dim = 100000;
                out.print("prova"+i+" ");
                //un test completo
                do{
                    out.print(" "+(new Sort(dim)).ordina());
                    Log.d("java", "prova "+i+", dim="+dim);
                    dim += 100000;
                }while(dim <= 2000000);
                out.println("");//new line per la nuova prova
            }//for

            //chiusura stream del file
            out.flush();
            out.close();
            fos.flush();
            fos.close();
        }catch(Exception e){
            Log.e("QuickSortJava","Errore nella gestione del
               file");
        }//catch
    }//onResume
}//QuickSortActivity
```

Classe Sort

Questa classe si occupa di inizializzare e riempire con elementi pseudo-casuali e di effettuarne il quicksort.

```
public class Sort {

    double array[];

    public Sort(int dim){
        array = new double[dim];
        riempi();
    }//Sort

    private void riempi(){
        for(int i=0 ;i<array.length; i++){
            array[i] = 100* Math.random(); //valore random da
                0 a 100
        }//for
    }//riempi
```

```
    public long ordina(){
        long inizio = System.currentTimeMillis();
        quickSort (0, array.length-1);
        long fine = System.currentTimeMillis();
        return fine - inizio;
    }//ordina

    private int partizione(int left, int right){
        int i = left, j = right;
        double tmp;
        double pivot = array[(left + right) / 2];

        while (i <= j) {
            while (array[i] < pivot)
                i++;
            while (array[j] > pivot)
                j--;
            if (i <= j) {
                tmp = array[i];
                array[i] = array[j];
                array[j] = tmp;
                i++;
                j--;
            }//if
        }//while
        return i;
}//partizione

    private void quickSort(int left, int right) {
        int index = partizione(left, right);
        if (left < index - 1)
            quickSort(left, index - 1);
        if (index < right)
            quickSort(index, right);
    }//quickSort
}//Sort
```

Sorgente ibrido

Classe QuickSortCActivity

Questa classe chiama il metodo per effettuare il quicksorf tramite JNI, quindi deve anche preparare l'array per il passaggio e, ovviamente, gestire la scrittura dei tempi sul file.

```
import java.io.File;
```

```
import java.io.FileOutputStream;
import java.io.PrintStream;

import android.app.Activity;
import android.os.Bundle;
import android.os.Environment;
import android.util.Log;

public class QuickSortCActivity extends Activity {

    int dim;
    double a[];

    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }

    public void onResume(){
        super.onResume();
        try{
            //apertura file
            File file = new File(Environment.
               getExternalStorageDirectory() + File.separator
                + "quickC3.txt");
            file.createNewFile();
            FileOutputStream fos = new FileOutputStream(file)
               ;
            PrintStream out = new PrintStream(fos);
            //10 prove per qualunque valore
            for(int i=0; i<10; i++){
                dim = 100000;
                out.print("prova"+i+" ");
                //un test completo
                do{
                    riempi();
                    out.print(" "+ sort(a, dim));
                    Log.d("native", "prova "+i+", dim="+dim);
                    dim += 100000;
                }while(dim <= 2000000);
                out.println("");//new line per la nuova prova
            }//for
            //chiusura stream del file
            out.flush();
            out.close();
```

```
            fos.flush();
            fos.close();
        }catch(Exception e){
            Log.e("QuickSortJava","Errore nella gestione del
               file");
        }//catch

    }//onResume

    private void riempi(){
        a = null; //forzatura del garbage collector
        a = new double[dim];
        Log.d("native","array di "+dim+" elementi");
        for(int i=0 ;i<dim; i++){
            a[i] = 100*Math.random(); //valore random
        }//for
    }//riempi

    public native double sort(double array[], int dim);

    static {
        System.loadLibrary("QuickSortC");
    }//static
}//QuickSortCActivity
```

Modulo QuickSortC

Il modulo JNI di questo programma effettua il quicksort sull'array passato tramite riferimento, calcolando il tempo impiegato.

```
#include <string.h>
#include <jni.h>
#include <time.h>
#include <android/log.h>

double diff (timespec start, timespec end);
int partizione(double arr[], int left, int right);
void quickSort(double arr[], int left, int right);

 extern "C" {
     JNIEXPORT jdouble JNICALL
        Java_it_unipd_dei_tesi_QuickSortCActivity_sort(JNIEnv
         * env, jobject obj, jdoubleArray array, jint dim);
 };
```

```
JNIEXPORT jdouble JNICALL
   Java_it_unipd_dei_tesi_QuickSortCActivity_sort(JNIEnv *
   env, jobject obj, jdoubleArray array, jint dim) {
    timespec start, end;
    int i; //indici
    jdouble *a; //array di copia
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &start);//tempo
       di inizio del programma

    a = env->GetDoubleArrayElements(array, JNI_FALSE);
    int d = dim;
    quickSort(a, 0, d-1);
    env->ReleaseDoubleArrayElements(array, a, JNI_ABORT);

    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &end);//tempo di
        fine del programma
    return diff(start, end);
}//sort

/*Funzione per il calcolo della differenza tra due timespec
   */
double diff (timespec start, timespec end){
   timespec tempo;
   if ((end.tv_nsec-start.tv_nsec) < 0){ //correzione per un
       giro di orologio
       tempo.tv_sec = end.tv_sec-start.tv_sec-1;
       tempo.tv_nsec = 1000000000+end.tv_nsec-start.tv_nsec;
   }else{
       tempo.tv_sec = end.tv_sec-start.tv_sec;
       tempo.tv_nsec = end.tv_nsec-start.tv_nsec;
   }//else
   return (double)((double)tempo.tv_sec*1000 + (double)tempo
      .tv_nsec/1000000);
}//diff


/*Funzioni che ordinano l'array*/
int partizione(double arr[], int left, int right){
   int i = left, j = right;
   double tmp;
   double pivot = arr[(left + right) / 2];

       while (i <= j) {
           while (arr[i] < pivot)
               i++;
           while (arr[j] > pivot)
               j--;
```

```
            if (i <= j) {
                tmp = arr[i];
                arr[i] = arr[j];
                arr[j] = tmp;
                i++;
                j--;
            }//if
    }//while
    return i;
}//partizione

void quickSort(double arr[], int left, int right) {
    int index = partizione(arr, left, right);
    if (left < index - 1)
        quickSort(arr, left, index - 1);
    if (index < right)
        quickSort(arr, index, right);
}//quickSort
```