Università degli Studi Roma Tre
Corso di Laurea Magistrale in Matematica

Algoritmi per il calcolo di radici quadrate nei campi finiti su risorse di calcolo massivamente parallele

Relatore:
Roberto Di Pietro, PhD

Candidato :
Laila Akif

Correlatore:
Flavio Lombardi, PhD

Anno Accademico 2013/2014

Sintesi

Il Problema

Negli ultimi anni cresce sempre di più il bisogno di risolvere nel tempo più breve possibile problemi che richiedono grandi risorse di calcolo. Alcuni di questi problemi scientifici sono l'elaborazione di immagini, come ad esempio quelle mediche, di ricerca geologica, o l'analisi di modelli di rischio finanziari; altri problemi riguardano la sicurezza informatica come la confidenzialità dei dati, e metodi contro attacchi DoS [11]. I sistemi crittografici attuali si basano sulla difficoltà computazionale dei problemi matematici, e si avvalgono di sistemi informatici e di programmi che svolgono complesse operazioni matematiche. Il calcolo delle radici quadrate modulo un numero primo è una delle operazioni matematiche che si potrebbe velocizzare su risorse di calcolo massivamente parallele.

Scenario Tecnologico

Negli ultimi anni la facilità di accedere a macchine che permettono un reale calcolo parallelo è aumentata, in particolare ora è facile disporre di macchine con processori multicore e con schede grafiche programmabili; CPU multicore sono poi contenute anche in personal computer a basso costo e dispositivi mobili, e le GPU non sono più da tempo relegate al solo calcolo dell'output grafico. L'uso delle GPU per effettuare calcoli paralleli viene chiamato GPGPU (General Purpose Computing on GPU) [13]. La GPU (Graphics Processing Units) indica l'unità di elaborazione grafica, cioè la componente centrale di una scheda video. Entrambi i maggiori produttori di GPU (ATI e Nvidia) hanno sviluppato gli strumenti necessari per sviluppare applicazioni General purpose sfruttando la potenza di calcolo delle GPU. La Nvidia Corporation fornisce un Software Development Kit (SDK) per la scrittura di programmi in grado di sfruttare le sue GPU basate sulla Compute Unified Device Architecture (CUDA) per calcoli non necessariamente legati alla grafica. Invece OpenCL (Open Computing Language) nasce da un'idea di

Apple che pensa di creare un linguaggio standard per il calcolo parallelo e di tipo general purpose. La prima specifica di OpenCL(1.0) viene rilasciata nel 2008 da Kronos Group, un consorzio che riunisce tutte le principali aziende del settore della Computer graphics (da Nvidia ad AMD/ATI).

Motivazione ed Obiettivi della Tesi

Lo scopo di questo lavoro è quello di utilizzare OpenCL per lo sviluppo di una versione parallela dell'algoritmo che calcola le radici quadrate nei campi finiti, l'algoritmo di Shanks Tonelli. Implementeremo gli algoritmi di Shanks Tonelli e di SZE in C, e mostreremo come utilizzare OpenCL per realizzare un'implementazione parallela più efficiente della versione sequenziale dell'algoritmo di Shanks Tonelli.

Nel capitolo 1 introduciamo delle nozioni matematiche che serviranno per la descrizione degli algoritimi che studieremo.

Definizione 1 Sia p un primo dispari e a un intero tale che $p \not| a$. Se la congruenza

$$x^2 \equiv a \, (mod \; p)$$

è risolubile, allora a si dice residuo quadratico di p, altrimenti a si dice non residuo quadratico di p.

Proposizione 1 Sia p un primo dispari che non divide a. Se a è un residuo quadratico di p, allora

$$a^{\frac{p-1}{2}} \equiv 1 \, (\text{mod} \;\; p)$$

Se invece a è un non redsiduo quadratico di p, allora

$$a^{\frac{p-1}{2}} \equiv -1 \, (\text{mod} \;\; p).$$

Un criterio per veificare se un elemento sia un residuo quadratico o no, più semplice di quello di Eulero si ottiene a mezzo della legge di reciprocità quadratica.

Teorema 1[12, Cap.3, §7] Legge di reciprocità quadratica
Siano p e q primi dispari distinti. Allora

$$\left(\frac{p}{q}\right) = \left(\frac{q}{p}\right)$$

a meno che sia $p \equiv q \equiv 3 \pmod 4$, nel qual caso

$$\left(\frac{p}{q}\right) = -\left(\frac{q}{p}\right).$$

Definizione 2 (Il problema delle radici q-esime) Sia G un gruppo ciclico finito e p un primo tale che $q^2|ord(G)$. Se w, a sono elementi in G tale che $a^q = w$ allora a è la radice q-esima di w. Trovare la radice q-esima a dato $w \in G$ è detto problema della radice q-esima.

Nel capitolo 2 descriviamo una possibile applicazione dei due algoritmi di Shanks Tonelli e di Sze nel campo della sicurezza; esponiamo in dettaglio l'algoritmo di Shanks-Tonelli e proponiamo una pseudocodifica della versione parallela di questo. Una volta esposto l'isomorfismo di SZE [20] su cui si basa l'algoritmo omonimo, descriviamo in dettaglio tale algoritmo.

La crittografia pubblica è una branca relativamente nuova della crittografia: i primi crittosistemi furono pubblicati nel 1970. Nonostante la sua giovinezza, gran parte della nostra moderna infrastruttura informativa per la sicurezza efficiente si basa sulla crittografia a chiave pubblica. I crittosistemi sono costruiti su problemi matematici computazionalmente impossibili; I due problemi più comuni sono la fattorizzazione di interi composti e il calcolo del logaritmo discreto. È noto che questi due problemi sono computazionalmente equivalenti, ma è meno nota l'altra equivalenza computazionale tra i problemi del logaritmo discreto e il calcolo delle radici q-esime. Il calcolo delle radici q-esime mod n è praticamente impossible, mentre è relativamente facile calcolare radici quadrate modulari mod p e mod q e calcolare da questi valori una radice quadrata mod n. Questo ultimo caso è quello di cui ci occupiamo; vediamolo più nel dettaglio.

Dato un primo dispari p, è facile vedere che per un dato intero a , la congruenza

$$x^2 \equiv a \pmod p$$

può non avere soluzioni (in questo caso diremo che a non è un residuo quadratico), una soluzione se $a \equiv 0 (\text{mod } p)$, oppure due soluzioni(a è residuo quadratico). Definendo il simbolo di Legendre $(\frac{a}{b})$ come -1 se a non è un residuo quadratico, 0 se a=0, e 1 se a è un residuo quadratico, abbiamo che il numero delle soluzioni mod p della congruenza è $1 + (\frac{a}{b})$. Il simbolo di Legendre è fondamentale in molti problemi, dunque si ha bisogno di un modo per calcolarlo; l'algoritmo 1.2.1 [4] usa la congruenza $a^{\frac{p-1}{2}} \equiv (\frac{a}{b}) \text{mod } p$. (possiamo migliorare questo algoritmo usando la legge di reciprocità quadratica di Legendre-Gauss).

Supponendo che la congruenza di sopra abbia soluzioni, abbiamo bisogno di un modo veloce per trovarle. Se $p \equiv 3 \pmod 4$ la soluzione è data da

$$x = a^{\frac{p+1}{4}} \pmod p$$

il calcolo della potenza viene fatto utilizzando l'algoritmo 1.2.1 [4]. Infatti, siccome a è un residuo quadratico, abbiamo $a^{\frac{p-1}{2}} \equiv 1 \pmod p$, quindi

$$x^2 \equiv a^{\frac{p+1}{2}} \equiv a.a^{\frac{p-1}{2}} \equiv a \qquad \pmod p$$

Per i restanti primi, cioè per i primi p tale che $p \equiv 5 \pmod 8$, abbiamo una soluzione meno banale. Siccome $a^{\frac{p-1}{2}} \equiv 1 \pmod p$ e $\mathbb{F}_p = \mathbb{Z}/p\mathbb{Z}$ è un campo, abbiamo

$$a^{\frac{p-1}{4}} \equiv \pm 1 \pmod p.$$

Ora, se il segno è +, allora si può facilmente verificare che

$$x = a^{\frac{p+3}{8}} \pmod p$$

è una soluzione. Altrimenti, dato che $p \equiv 5 \pmod 8$ applicando la legge di reciprocità quadratica, sappiamo che $2^{\frac{p-1}{2}} \equiv -1 \pmod p$. Allora possiamo verificare che

$$x = 2a(4a)^{\frac{p-5}{8}} \pmod p$$

è una soluzione.

Infine l'ultimo caso è $p \equiv 1$ $(mod\ 8)$ che viene risolto dai due algoritmi che ora vediamo.

L'algoritmo di Shanks-Tonelli calcola le radici quadrate dei numeri interi modulo un numero primo, quindi le radici quadrate nel campo finito $\mathbb{Z}/p\mathbb{Z}$, dove p è un numero primo. L'algoritmo pubblicato da Daniel Shanks nel 1972 è simile ma più efficiente di un altro algoritmo descritto da Alberto Tonelli nel 1891. Calcolare le radici quadrate in $\mathbb{Z}/2\mathbb{Z}$ è banale, quindi consideriamo d'ora in avanti solo il caso p primo dispari.

Sia z un generatore del gruppo moltiplicativo G di $\mathbb{Z}/p\mathbb{Z}$, dato che p è un numero dispari, possiamo sempre scrivere $p - 1 = 2^e q$,con q dispari. Allora, se a è un residuo quadraticomod p, si ha

$$a^{\frac{p-1}{2}} = (a^q)^{2^{e-1}} \equiv 1 \pmod p;$$

$b = a^q \bmod p$ è una radice in $\mathbb{G}$, dunque esiste un intero k con $0 \leq k < 2^e$ tale che:

$$a^q z^k = 1$$

in $\mathbb{G}$; inoltre se si pone $x = a^{\frac{q+1}{2}} z^{\frac{k}{2}}$, è chiaro che $x^2 \equiv a(\text{mod } p)$, quindi x è la radice cercata.

Quindi, per ottenere un algoritmo dobbiamo risolvere due problemi :

- Trovare un generatore z di $\mathbb{G}$.
- Calcolare l'esponente k.

Sebbene molto facile da risolvere, il primo problema è la parte probabilistica dell'algoritmo. La migliore strada per trovare z è la seguente:
Scegliere un intero n random, e calcolare $z = n^q \bmod p$. Dunque z è un generatore di $\mathbb{G}$ se e solo se n è un non residuo quadratico mod p (questo si verifica con probabilità esattamente $\frac{p-1}{2p}$) che si verifica facilmente. Trovare l'esponente k è abbastanza difficile, ma osserviamo che non se ne ha bisogno esplicitamente.
Diamo una descrizione dell'algoritmo di Shanks, il quale calcola la radice quadrata in tempo polinomiale [2].

Descrizione Algoritmo:

Dati p primo, tale che $p = 2^e q + 1$, e un residuo quadratico a e un non residuo quadratico n. Possiamo calcolare x tale che $x^2 \equiv a \,(\text{mod } p)$. L'algoritmo funziona come segue:

Algoritmo:

1. Si fissi $r = e$, $g = n^q$, $x = a^{\frac{q+1}{2}}$, $b = a^q$.
2. Sia m il minimo intero tale che $b^{2^m} \equiv 1 \,(\text{mod } p)$.
3. Si ponga $t = g^{2^{r-m-1}}$, $g = t^2$,$b = bg$, $x = xt$.
4. Se $b = 1$ Stop e return 1, altrimenti poni $r = m$ e vai al passo 2.

Algorithm 1 Tonelli-Shanks

Data: $\mathbb{F}_p$, p primo dispari , $a \in \mathbb{Z}$;
$p - 1 = 2^e q$ q dispari;
Result: x **:** $x^2 \equiv a \,(\mathbf{mod}\ \ p)$;
---
n random : $(\frac{n}{p}) = -1$;
$y \leftarrow n^q \,(\text{mod}\ \ p)$;
$r \leftarrow e$;
$x \leftarrow a^{\frac{q+1}{2}} \,(\text{mod}\ \ p)$;
$b \leftarrow ax^2 \,(\text{mod}\ \ p)$;
$x \leftarrow ax \,(\text{mod}\ \ p)$;
while $m \neq 0$ **do**
  if $b \equiv 1 \,(\text{mod}\ \ p)$ **then**
    return x
  else
    Trova il più piccolo $m \geq 1 : b^{2^m} \equiv 1 \,(\text{mod}\ \ p)$;
  end if
  if $m \equiv r \,(\text{mod}\ \ p)$ **then**
    return a è un non-residuo quadratico mod p
  end if
  $t \leftarrow y^2$;
  $y \leftarrow t^2$;
  $r \leftarrow m$;
  $x \leftarrow xt$;
  $b \leftarrow by$;
end while
return x

Descriviamo una versione parallela dell'algorimo:

Algoritmo parallelo:

1. Si fissi $r = e$, $g = n^q$, $x = a^{\frac{q+1}{2}}$,$b = a^q$.
2. Si calcoli g^2, g^{2^2}, g^{2^3}, ..., g^{2^e} e salva questi valori in un array.
3. Si calcoli b^2, b^{2^2}, b^{2^3}, ...,b^{2^e} e salva questi valori in un array.
4. Trovi il più piccolo intero m, tale che $b^{2^m} \equiv 1 \pmod{p}$.
5. Si Ponga $t = g^{2^{r-m-1}}$, $g = t^2$, $x = xt$, $r = m$.
6. Si ponga $b = bg$,calcola b^2, b^{2^2}, b^{2^3}, ...,b^{2^e} e aggiorna il vettore delle potenze di b.
7. Se $b = 1$, stop e return x, altrimenti vai al passo 4.

Tutti i passi nella versione parallela dell'algoritmo appena descritta ad eccezione del 6, possono essere eseguiti da un singolo processore. Il punto 6 deve essere eseguito al più r volte e si può eseguire in parallelo da r processori in un unico passaggio. Infatti, poiché i vecchi valori di b sono memorizzati, i nuovi valori di b si ottengono con una semplice moltiplicazione, e tutte queste moltiplicazioni possono essere eseguite indipendentemente una dall'altra su differenti processori. Quindi questo algoritmo viene eseguito in tempo $O(\log q + n)$ su r processori.

Descriviamo ora l'algoritmo di SZE:
L'idea di Sze consiste nel ridurre il problema del calcolo della radice quadrata di β al problema della costruzione della r-esima radice primitiva dell'unità $\zeta_r \in F_q$ per qualche r tale che $r \mid q - 1$. L'ingrediente principale di questa riduzione è un isomorfismo di gruppi che descriviamo. Consideriamo il campo finito con q elementi $\mathbb{F}_q$, supponiamo che $\beta \in \mathbb{F}_q^*$ sia una radice quadrata. Allora

$$\alpha^2 = \beta \, per \; qualche \; \alpha \in \mathbb{F}_q$$

Definiamo un gruppo G_α con le seguenti proprietà:

- l'operazione del gruppo in G_α è efficientemente calcolabile conoscendo β ma non α,
- G_α è isomorfo al gruppo moltiplicativo $\mathbb{F}_q^*$,
- l'isomorfismo $\psi : G_\alpha \longrightarrow \mathbb{F}_q^*$ dipende dal parametro α.

- $G_\alpha := G'_\alpha \cup \{\zeta\}$, dove $G_\alpha{}' = \{[a] : a \in F_q, a \neq \pm\alpha\}$ e ζ è l'elemento neutro.

Definiamo ora un'operazione $*$ su G_α: per ogni $[a] \in G_\alpha$ e $\forall [a_1], [a_2] \in G'_\alpha$ con $a_1 + a_2 \neq 0$,

$$[a] * \zeta = \zeta * [a] = [a],$$

$$[a_1] * [-a_2] = \zeta,$$

$$[a_1] * [a_2] = \Big[\frac{a_1 a_2 + \alpha^2}{a_1 + a_2}\Big].$$

$(G_\alpha, *)$ è un gruppo ben definito ed è isomorfo al gruppo moltiplicativo ${F_q}^*$. Useremo G_α per costruire l'algoritmo deterministico per il calcolo della radice quadrata. **Proposizione 2** Sia $[b] \in G'_\alpha$. Per $d > 0$, risulta $[b]^d = \zeta$ se e solo se $\psi_d(b) = 0$.

Proposizione 3 Se $b \in G_\alpha$ tale che $b^2 \neq \zeta$ e se l'ordine di b in $(G_\alpha, \star)$ è d, allora

$$\alpha = b\big(\frac{{\zeta_d}^k + 1}{\zeta_d^k - 1}\big) \qquad per \quad 0 < k < d, \quad k \neq \frac{d}{2}$$

dove ζ_d è una radice d-esima dell'unità.

La proposizione appena enunciata suggerisce un metodo per calcolare la radice quadrata α, conoscendo un elemento $[b] \in G_\alpha$ di ordine d, cioè una radice d-esima dell'unità $\zeta_d \in F_q^*$, e l'indice k. Data l'equazione risolvibile $\alpha^2 = \beta$ in ${F_q}^*$, presentiamo l'algoritmo deterministico di Sze per il calcolo della radice quadrata su $\mathbb{F}_q$.

Descrizione Algoritmo: Scriviamo $q = 2^e {p_1}^{e_1} ... {p_n}^{e_n} t + 1$, dove $p_1, ..., p_n$ sono n primi distinti dispari e $t, e, e_1, ..., e_n$ interi positivi tale che $(2p_1...p_n, t) = 1$. Supponiamo $e > 1$, altrimenti il problema è banale.

Algorithm 2 SZE

Data: β residuo quadratico;
q primo dispari;
$g_1, g_2, ..., g_{2t-1} \in {F_q}^* : 2t-1$ elementi distinti in ${F_q}^*$
Result: $\pm\sqrt{\beta}$;
---
if $\exists j \in [1, 2t-1] : {g_j}^2 = \beta$ **then**
  return $\pm g_j$
else
  $g \leftarrow g_k : g_k^{2t} \neq \zeta$ per qualche $k \in [1, 2t-1]$;
end if
if $g^{\frac{q-1}{2^{e-1}}} \neq \zeta$ **then**
  while $k < e$ **do**
    $g^{\frac{q-1}{2^k}} = \zeta$;
    $k++$;
  end while
  $a \leftarrow [g]^{\frac{q-1}{2^{k+2}}}$;
  return $\pm a\sqrt{-1}$
else
  while $m < n$ e $[g]^{(q-1)/{p_m}^e_m} = \zeta$ **do**
    m++;
  end while
  if $m < n$ **then**
    $r \leftarrow p_m$;
  end if
  for $k \in \{1, e\}$ **do**
    if $g^{\frac{q-1}{r^k}} \neq \zeta$ **then**
      return $k-1$
    end if
  end for
  $a \leftarrow g^{\frac{q-1}{r^{k+1}}}$;
  $\zeta \leftarrow \zeta_r \in F_q$;
  while $j < \frac{r-1}{2}$ **do**
    if $(a(\zeta_r^j - 1)/(\zeta_r^j + 1))^2 = \beta$ **then**
      return j
    end if
    $j++$;
  end while
end if
return $\pm a(\zeta_r^j - 1)/(\zeta_r^j + 1)$

In entrambi gli algoritmi che abbiamo descritto sono presenti diverse istruzioni di cicli e funzioni che usano operazioni matematiche modulari complesse, che richiedono molto tempo se eseguite in modo seriale; per parallelizzare il codice abbiamo bisogno di strumenti per sviluppare il codice parallelo.

Nel capitolo 3 introduciamo il calcolo basato su GPU, il GPU computing, e l'ambiente di programmazione OpenCL per la programmazione di sistemi eterogenei. L'unità di elaborazione grafica, ovvero la GPU inizialmente presentata per migliorare il rendering grafico ed essenzialmente per velocizzare le operazioni più pesanti nell'ambito della grafica tridimensionale grazie all'elevato numero di core, viene utilizzata come coprocessore della CPU e da alcuni anni anche in generiche elaborazioni dati. OpenCL (Open Computing Language) è una libreria basata sul linguaggio ANSI C che può esser eseguito su una molteplicità di piattaforme, CPU, GPU, e altri tipi di processori. In particolare, le potenzialità di OpenCL sono ben espresse con architetture altamente parallelizzabili e potenti come le GPU, e in questo caso si parla dell'ambito GPGPU. L'espressione (GPGPU General Purpose Computation on Graphics Processing Units) indica la tendenza di sfruttare le GPU per calcoli non strettamente correlati ai loro compiti originari.

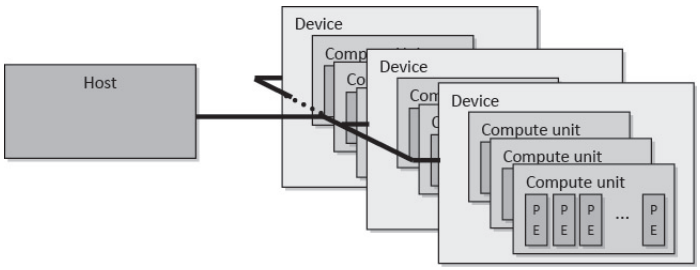

Figura 1: Modello della piattaforma in OpenCL.

Lo standard OpenCL è strutturato su tre livelli ognuno riguardante aspetti differenti come il linguaggio, il sistema a runtime, e la piattaforma. Gli elementi che lo compongono sono: Il modello della piattaforma che stabilisce in che modo le varie entità sono collegate tra loro; il modello d'esecuzione OpenCL, che definisce come avviene l'esecuzione dei kernel; i kernel, che sono le parti di un Programma OpenCL che vengono eseguite sui dispositivi; il modello della memoria, dove sono definite quattro tipi di memoria e descritta la gerachia tra queste; il contesto, che è creato dal lato host ed è costituito da da una collezione di dispositivi OpenCL, dai kernel, gli oggetti programma gli ogetti memoria; La coda dei comandi permette l'interazione tra host e device; il modello di programmazione che definisce due modelli di programmazione parallela, il parallelismo ripetto ai dati, e il parallelismo rispetto ai

task. Nella programmazione parallela lo sviluppatore deve scegliere il modello di programmazione e l'hardware parallelo adatto al problema; l'hardware è generalmente più adatto per alcuni livelli di parallelismo e meno per altri. I processori Multicore (CPU AMD e Intel multicore) sono adatti al parallelismo a livello task, mentre i processori Manycore SIMD (GPU di AMD e Nvidia) sono adatti al parallelismo rispetto ai dati.

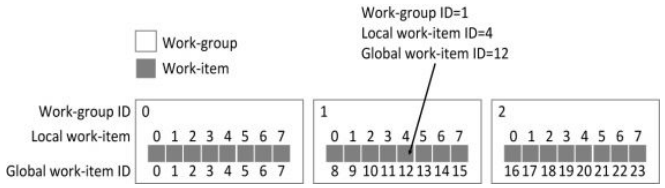

Figura 2: NDrange di dimensione N=1 in OpenCL.

Il modello d'esecuzione definisce come avviene l'esecuzione dei kernel. Quando un'applicazione lancia l'escuzione di un kernel su un device il runtime OpenCL crea un indice di spazio di interi sul quale verrà eseguito il kernel, su ogni elemento dell'indice viene eseguita un'istanza del kernel. Questa istanza è detta work-item ed è identificata da un ID globale che rappresenta le sue coordinate nell'indice spazio. L'indice di spazio è detto NDRange ed è N-dimensionale, dove N può essere 1, 2 o 3. Prima che i work-item siano eseguiti essi vengono raggruppati in work-group e identificati da un ID univoco. I work-group vengono distribuiti alle compute unit che ne eseguono i work-item presenti utilizzando i processing element.

Secondo la tassonomia di Flynn per le architetture, l'utilizzo di OpenCL ricade normalmente nel caso SPMD (Single Program Multiple Data) [15] per l'esecuzione generale del programma, l'hardware della GPU applica automaticamente gli work item di tipo SPMD sulle unità di tipo SIMD interne; questo viene definito a volte SIMT (Single Instruction Multiple Thread), quindi esistono esecuzioni SIMD in cui l'esecuzione complessiva rimane SPMD. Quindi l'approccio migliore per l'esecuzione OpenCL su GPU è di tipo SPMD (sugli work item) con decomposizione massiccia sui dati (in particolare di output per applicazioni scientifiche) ed esecuzione SIMD in ogni item. Per le CPU invece l'approccio migliore è un numero ridotto di thread utilizzando MIMD (Multiple Instruction Multiple Data) con decomposizione sulle attività o SPMD con decomposizione sui dati, utilizzando sempre un numero ridotto di thread.

Nel capitolo 4 descriviamo il codice OpenCL del progetto realizzato, una versione parallela dell'algoritmo di Shanks Tonelli. Nella versione precedente dell'algoritmo , il passo 6 dove aggiorniamo le potenze del fattore di correzione, ovvero il vettore delle potenze di b, viene eseguito generalmente al massimo r volte. Poiché in questo passo l'aggiornamento del vettore C è calcolato sulla base dei valori del precedente vettore C e dei valori del vettore B delle potenze di g (aggiornato nel passo precedente), abbiamo proposto una modalità di precompilazione di tutti i vettori B necessari per aggiornare i vettori C ad ogni passo, in modo da poter calcolare tutti i vettori C necessari all'esecuzione dell'algoritmo. Per maggiori chiarimenti proponiamo il seguente schema:

Algorithm 3 Tonelli-Shanks Lato host

1: Inizializza i parametri: calcola $r = e$, $g = n^q$, $x = a^{\frac{q+1}{2}}$,$b = a^q$;
2: Calcola g, g^2, g^{2^2}, g^{2^3}, ..., g^{2^e} e salva questi valori nell'array B^0;
3: Calcola b^2, b^{2^2}, b^{2^3}, ...,b^{2^e} e salva questi valori nell'array C,
4: Compila la matrice B per colonne, le successive colonne si ottengono elevando al quadrato la precedente colonna;
5: Copia la matrice B e il vettore C nella memoria del device;
6: Esegui il kernel, quindi calcola la matrice D, le cui colonne sono i vettori C aggiornati;
7: Leggi la matrice D dal device verso le memoria dell'host;
8: Trasponi la matrice D ed esegui un ciclo scorrendo le righe della matrice per calcolare la soluzione.

Calcola la matrice D nel seguente modo: per ogni j-esima colonna di D, esegui il prodotto elemento ad elemento della colonna C, la j-esima colonna di B, e le precedenti (j-1)-esime colonne di B; Il prodotto viene eseguito richiamando la funzione `molt_n` dal kernel, quest'ultima recupera i parametri necessari contenenti nei vettori B e C dalla memoria globale; anche l'output della stessa funzione viene copiato nella memoria globale.

Per realizzare un'applicazione OpenCL, come descritto nel capitolo utilizzi 3 è necessario un codice host, che è un normale programma C, che utilizza tipi e funzioni contenuti nella libreria *cl.h* per costruire i kernel e avviare i rispettivi work-item, e un codice della funzione kernel spostato nel file .cl
Possiamo vedere una Flowchart sintetica che ci permette di capire ancora meglio lo schema appena proposto:

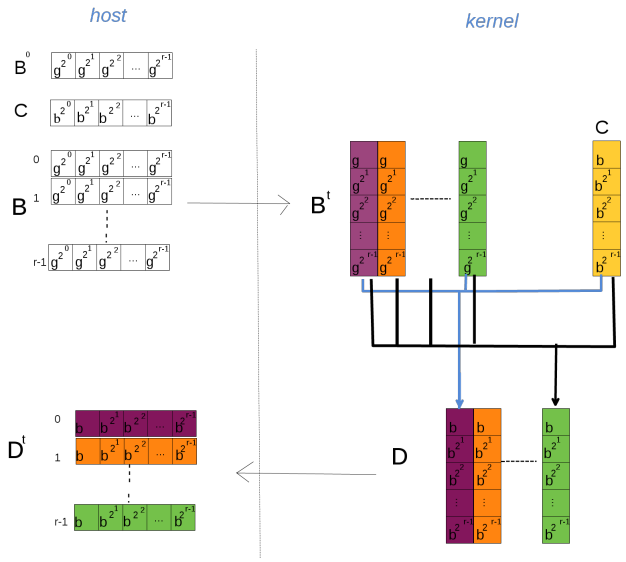

Figura 3: Flowchart della precompilazione della matrice delle potenze dei fattori di correzione.

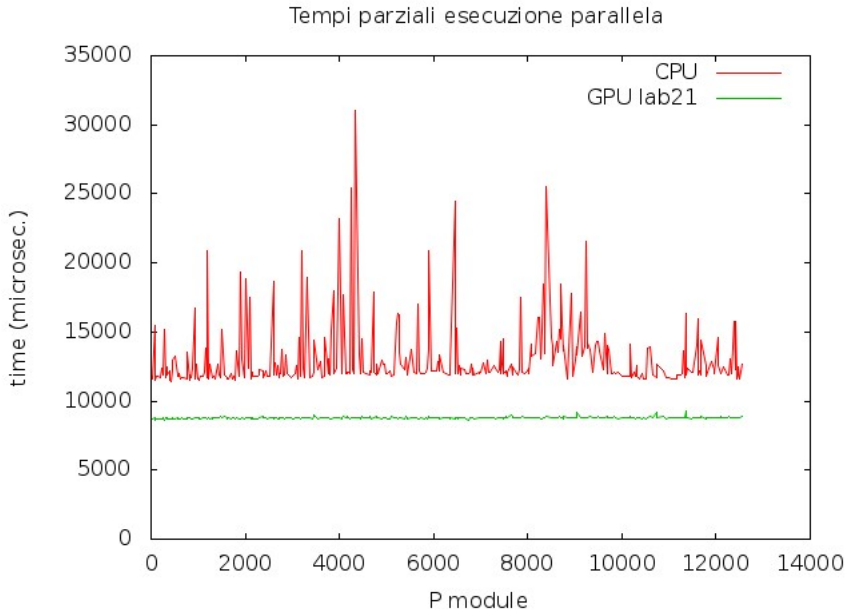

Figura 4: Tempo di esecuzione in microsecondi dell'esecuzione parallela di Shank-Tonelli su CPU e GPU lab21.

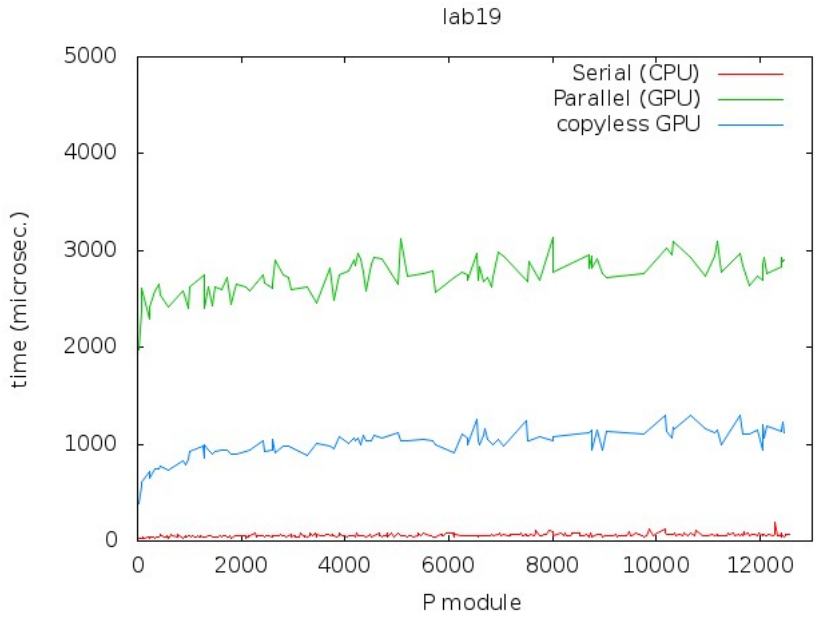

Figura 5: Tempi di esecuzione in microsecondi dell'esecuzione di Shanks-Tonelli seriale sulla CPU Pentium Dual-Core e della sua versione parallela sulla GPU AMD Radeon 6800 in lab19.

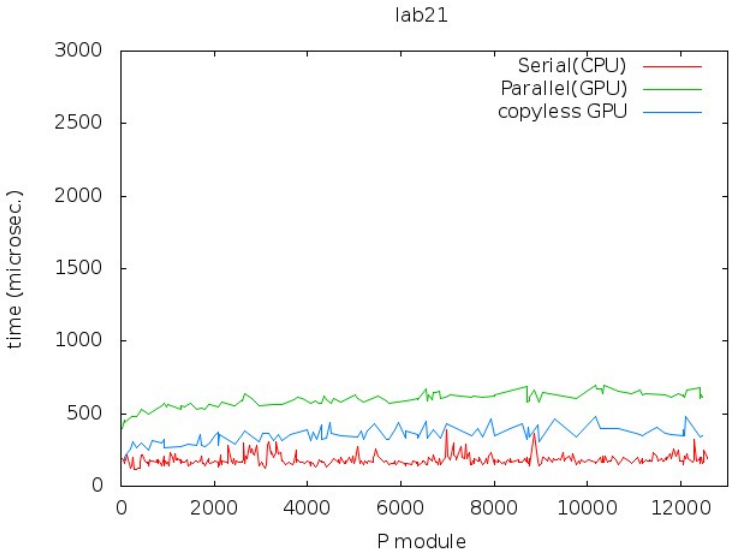

Figura 6: Tempi di esecuzione in microsecondi dell'esecuzione seriale sulla CPU Pentium Dual-Core e l'esecuzione parallela di Shank-Tonelli sulla GPU Tesla C2070.

Nella fase sperimentale, le analisi delle prestazioni degli algoritmi sono state svolte su diversi dispositivi forniti dall'università Roma Tre: Pentium(R) Dual Core, AMD Athlon(tm) II X4 640 Processor (che chiamaeremo CPU lab19); GPU Tesla C2070 Nvidia (GPU lab21) e la GPU Barts (GPU lab19). Per valutarne le performance abbiamo graficato i risultati ottenuti dalle implementazioni parallela e seriale dell'algoritmo di Shanks Tonelli. Le prestazioni sono strettamente legate all'hardware utilizzato; Osserviamo nel grafico 4 che la versione parallela dell'algoritmo di Tonelli-Shanks eseguita sulla GPU è vantaggiosa rispetto a quella sulla CPU, indipendentemente dalla grandezza del modulo p.

Con l'aiuto degli oggetti event in OpenCL abbiamo misurato i tempi dei trasferimenti dei dati relativi all'esecuzione, e abbiamo appreso che la maggior parte del tempo è spesa nel trasferimento dei dati dal lato host al dispositivo, e dalla lettura dei risultati dal dispositivo tramite i buffer. Inoltre il tempo per il trasferimento dei buffer è proporzionale alla loro dimensione, abbiamo quindi diminuito la dimensione del valore *global-size* a soli 256B, invece dei 4MB. Nei grafici 6 e 5, infatti abbiamo graficato anche i tempi parziali che non considerano i trasferimenti dei dati, e come possiamo vedere sono molto inferiori. Nonostante ciò, risulta più veloce l'esecuzione seriale piuttosto che quella parallela in tutte le tre macchine che abbiamo a disposizione.

Le mancate prestazioni a favore della versione parallela dell'algoritmo parallelo di Shanks-Tonelli sono dovute al continuo accesso alla memoria globale e al calcolo del prodotto modulare tramite la funzione `molt_n`. Infatti la funzione `molt_n` viene richiamata spesso da tutti i work item che eseguono il kernel, dunque recupera e scrive continuamente dati dalla memoria globale. Per migliorare le prestazioni dell'esecuzione parallela abbiamo diminuiamo il numero dei buffer da trasferire.

Implementiamo un nuovo kernel che fa un benchmark della funzione `molt_n`, e confrontiamo i risultati su diversi dispositivi. Per effettuare il test generiamo in maniera casuale 10000 numeri compresi in un intervallo controllato, li salviamo in un array, e infine eseguiamo il codice OpenCL con input l'array che ne calcola il prodotto modulare richiamando la funzione `molt_n` dal kernel.

Nella tabella 1 abbiamo riportato il tempo medio necessario per calcolare il prodotto modulare con la funzione `molt_n` sulle diverse macchine. I tempi riportati nella prima riga della tabella rispecchiano i risultati ottenuti nei grafici precedenti dell'algoritmo di Shanks-Tonelli: osserviamo infatti come sia molto più costoso calcolare il prodotto modulare sulle GPU di lab19 e lab21. Questa differenza nelle prestazioni è dovuta al recupero dei valori

Tempo medio Molt_n in nanosecondi				
	CPU lab19 AMD X4	**GPU lab21 Tesla C2070**	**GPU lab19 AMD Radeon6800**	**CPU serial Pent. Dual-Core**
serial -global mem.	107 ns	8569 ns	13700 ns	59 ns
serial-copyless outp.	3 ns	1379 ns	26 ns	–
parallel-global mem.	3 ns	2 ns	31 ns	–
parallel-local mem.	3 ns	2 ns	29 ns	–

Tabella 1: Tempo medio del calcolo del prodotto modulo n

dalla memoria globale e al caricamento dei risultati. Nella seconda riga della tabella riportiamo i tempi ottenuti modificando il kernel e copiando i risultati dei prodotti nella memoria privata, dunque non caricando questi nella memoria globale i tempi diminuiscono esponenzialmente.

Nella terza riga della tabella sono riportati i tempi di esecuzione ottenuti sfruttando il calcolo parallelo. L'esecuzione in lab21 è più veloce poiché dipende dalle prestazioni dell'hardware, infatti la GPU Nvidia ha più unità di calcolo rispetto alla GPU AMD, quindi vengono eseguiti in parallelo un numero di work item maggiore nella GPU Nvidia.

Nell'ultima riga della tabella vediamo i tempi dell'esecuzione parallela con l'uso della memoria locale. Utilizzare la memoria locale inizializzata da un certo work item con i valori necessari allo stesso riduce il numero di accessi alla memoria globale alla sola copia iniziale e non al numero di occorrenze della variabile. Dai tempi riportati nella tabella non si evincono benefici nell'utilizzo della memoria locale poiché ogni work group ha un solo work item, e una volta che questo copia la variabile necessaria nella memoria locale la richiama poi una sola volta.

Dopo aver diminuito il numero dei buffer ci concentriamo sull'ottimizzazione del codice *kernel* dell'algoritmo e l'uso della memoria globale. Durante l'esecuzione dei work item, calcoliamo le colonne della matrice D in parallelo e ne calcoliamo al massimo r colonne. Al crescere del valore di r cresce il tempo di esecuzione del kernel, dunque modifichiamo il codice sorgente del kernel per calcolare solo le colonne necessarie al calcolo della radice. La figura 7 mostra il confronto tra la versione precedente dell'algoritmo parallelo e quella nuova con il kernel ottimizzato; osserviamo che l'esecuzione parallela è più vantaggiosa di quella seriale.

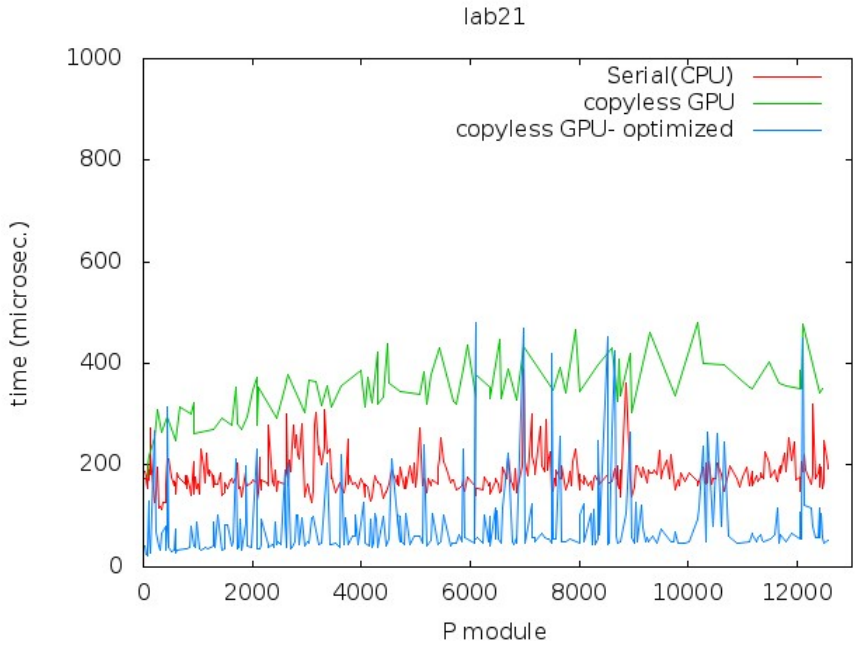

Figura 7: Tempo di esecuzione in microsecondi dell'esecuzione parallela di Shank-Tonelli sulla macchina lab21 con kernel ottimizzato.

Nonostante gli ultimi risultati che abbiamo ottenuto a favore dell'esecuzione parallela sulla GPU, ci sono diversi problemi già accennati che potrebbero essere studiati e risolti: la gestione esplicita dei buffer che permette di limitare un numero eccessivo di trasferimenti di dati tra CPU e GPU ma aumenta la difficoltà della gestione di questi e la leggibilità del codice OpenCL, l'uso della memoria globale, e la gestione di più kernel poiché questi non condividono la memoria.

Discussione e conclusioni

CPU e GPU hanno architetture e funzionamenti molto diversi, ed entrambe eccellono in diversi contesti. Ovviamente le CPU continuano ad essere le principali unità di computazione di qualsiasi sistema, ma la continua evoluzione delle GPU sta rivelando il grande aiuto che questi processori possono portare nei calcoli complessi non relativi a problemi grafici. Infatti con la nascita del GPGPU, i programmatori hanno avuto vantaggi sempre crescenti dall'utilizzo delle GPU, potendo migliorare una grande varietà di algoritmi tramite la parallelizzazione. Per rendere le GPU facilmente programmabili i produttori hanno sviluppato librerie del calcolo parallelo come CUDA da Nvidia, e OpenCL dal Kronos Group.

Nel nostro lavoro abbiamo cercato di migliorare l'utilità delle GPU nel campo della sicurezza, mostrando in particolare che grazie a questa componente hardware si possono migliorare le prestazioni del crittosistema di Rabin. Abbiamo studiato il problema delle radici quadrate nei campi finiti e le relative soluzioni a questo problema date dagli algoritmi di Shanks Tonelli e di SZE nel caso di residui quadratici modulo una potenza di 2. Abbiamo studiato i due algoritmi sottolineando gli aspetti matematici necessari alla loro implementazione e alla necessità di ottenere i risultati in un tempo minore. Grazie all'uso di OpenCL abbiamo parallelizzato l'algoritmo di Shanks Tonelli ed eseguito questa applicazione sulla CPU e sulle GPU dei due produttori Nvidia e AMD. Raccolti i risultati dei test ottenuti, li abbiamo confrontati con quelli ottenuti dall'implementazione seriale dello stesso algoritmo; i risultati ci hanno permesso di apprendere informazioni sui vantaggi, benefici e accorgimenti di questo approccio. In particolare ci siamo concentrati sull'ottimizzazione del codice parallelo per migliorare i tempi di esecuzione, infatti abbiamo creato un benchmark che calcola le prestazioni dei vari dispositivi eseguendo la funzione che limita le prestazioni dell'algoritmo; dal benchmark, come già dai risultati precedenti, abbiamo appreso che alcuni degli svantaggi dell'utilizzo della GPU sono il tempo di trasferimento dei dati dalla CPU alla GPU, la latenza alta a causa della cache limitata, e il costoso accesso alla memoria globale. Tuttavia le GPU, grazie alla moltitudine di core disponibili e al throughput elevato, riescono ad avere performance notevoli nel calcolo parallelo e, grazie alle istruzioni SIMD, riescono ad eseguire operazioni complesse in un numero di clock inferiore rispetto alle CPU, perciò la parallelizzazione è molto utile per migliorare le prestazioni. In conclusione abbiamo raccolto una quantità di dati che possono essere utilizzati come punto di partenza per parallelizzare l'algoritmo di SZE, partendo anche dalla versione seriale sviluppata.

Lavori futuri

Le GPU incrementeranno sempre di più il divario con le CPU per quello che riguarda la capacità di calcolo. Inoltre il sempre maggiore interesse degli sviluppatori verso la computazione eterogenea, favorirà l'orientamento delle architetture delle GPU al GPGPU, che anno dopo anno stanno facendo progressi non indifferenti. La Heterogeneous System Architecture (HSA) Foundation capitanata da AMD, nel 2013 [14] ha stabilito nuove specifiche per un'architettura di memoria condivisa, con i vari elementi computazionali (CPU, APU o GPU) presenti su un sistema informatico capaci di accedere indistintamente all'intero spazio di memoria installato sulla macchina. Queste specifiche permetteranno di superare gli svantaggi legati al trasferimento dei dati da CPU a GPU e all'accesso alla memoria globale. La nuova architettura di memoria si chiama Heterogeneous Unified Memory Access o HUMA, e sposta la gestione degli indirizzi e puntatori di memoria dal software al livello più basso dell'hardware dell'architettura PC.

Bibliografia

[1] Cheryl L. Beaver, Peter Gemmell, Anna M. Johnston, and William D. Neumann. On the cryptographic value of the qth root problem. In *Proceedings of the Second International Conference on Information and Communication Security*, ICICS '99, pages 135–142, London, UK, UK, 1999. Springer-Verlag.

[2] JEREMY BOOHER. Square roots in finite fields and quadratic nonresidues. 2012.

[3] Giulia Maria Piacentini Cattaneo. *Algebra: un approccio algoritmico.* Decibel, 1996.

[4] Henri Cohen. *A course in computational algebraic number theory*, volume 138. Springer, 1993.

[5] Benedict Gaster, Lee Howes, David R Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL: Revised OpenCL 1.* Newnes, 2012.

[6] Pascal Giorgi, Thomas Izard, Arnaud Tisserand, et al. Comparison of modular arithmetic algorithms on gpus. In *ParCo'09: International Conference on Parallel Computing*, 2009.

[7] Darrel Hankerson, Alfred J Menezes, and Scott Vanstone. *Guide to elliptic curve cryptography.* Springer, 2004.

[8] A. Hariri, K. Navi, and R. Rastegar. A simplified modulo (2n-1) squaring scheme for residue number system. In *Computer as a Tool, 2005. EUROCON 2005.The International Conference on*, volume 1, pages 615–618, 2005.

[9] Y.I. Jerschow and M. Mauve. Non-parallelizable and non-interactive client puzzles from modular square roots. In *Availability, Reliability*

and Security (ARES), 2011 Sixth International Conference on, pages 135–142, 2011.

[10] Anna Johnston. *On the Difficulty of Prime Root Computation in Certain Finite Cyclic Groups*. PhD thesis, University of London, 2006.

[11] Jing Yang Koh, J.T.C. Ming, and D. Niyato. Rate limiting client puzzle schemes for denial-of-service mitigation. In *Wireless Communications and Networking Conference (WCNC), 2013 IEEE*, pages 1848–1853, 2013.

[12] A. Languasco and A. Zaccagnini. *Introduzione alla crittografia: algoritmi, protocolli, sicurezza informatica*. Hoepli informatica. Hoepli, 2004.

[13] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. Gpgpu: general purpose computation on graphics hardware. In *ACM SIGGRAPH 2004 Course Notes*, SIGGRAPH '04, New York, NY, USA, 2004. ACM.

[14] Daniel Lustig and Margaret Martonosi. Reducing gpu offload latency via fine-grained cpu-gpu synchronization. In *High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on*, pages 354–365, Feb 2013.

[15] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[16] Benjamin J Rapone. Taking square roots over finite fields. 2012.

[17] Matthew Scarpino. *OpenCL in Action: how to accelerate graphics and computation*. Manning, 2012.

[18] René Schoof. Elliptic curves over finite fields and the computation of square roots mod p. *Mathematics of computation*, 44(170):483–494, 1985.

[19] Tsz-Wo Sze. On taking square roots without quadratic nonresidues over finite fields. *Mathematics of Computation*, 80(275):1797–1811, 2011.

[20] T.W. Sze and College Park. Computer Science University of Maryland. *On Solving Univariate Polynomial Equations Over Finite Fields and Some Related Problems*. University of Maryland, College Park, 2007.

[21] Gonzalo Tornaría. Square roots modulo p. In *Proceedings of the 5th Latin American Symposium on Theoretical Informatics*, LATIN '02, pages 430–434, London, UK, UK, 2002. Springer-Verlag.