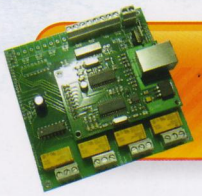
ioPROGRAMMO

PER ESPERTI E PRINCIPIANTI

Poste Italiane S.p.A Spedizione in A.P. • D.L. 353/2003 (conv.in L.27/02/2004 n.46) art.1 comma 2 DCB ROMA • Periodicità mensile • MARZO 2009 • ANNO XIII, N.3 (136)



DOMOTICA: UNA SCHEDA CHE SI CONTROLLA DAL WEB
Hardware programmabile e a basso costo per controllare da Internet i tuoi elettrodomestici

UN PC DA FARE nVIDIA®

LA POTENZA DELLE SCHEDE GRAFICHE PER METTERE LE ALI AL TUO SOFTWARE

- ✔ CUDA: la tecnologia per scrivere codice ultraveloce
- ✔ Utilizzare le API per modificare applicazioni già esistenti
- ✔ Installazione e configurazione dell'SDK in Visual Studio
- ✔ Gli esempi pratici pronti all'uso. A prova di cronometro!

TUTTO IL CODICE Sul CD

DATABASE

UN SOLO CODICE PER ACCEDERE A MILLE DB
Con LINQ interfacci le tue applicazioni a qualsiasi base dati

DAI FEED RSS AL TUO DATABASE
Un progetto in VB.NET per salvare dati RSS di un sito web in un DB

MOBILE

UN TOMTOM NEL TUO CELLULARE
Poche mosse per portare nel telefonino un navigatore GPS

PROGETTARE UN VIDEOGAME IN J2ME
Individuare l'input dell'utente e azionare i relativi eventi di gioco

ECLIPSE

PERFETTO ANCHE PER IL MOBILE
Come utilizzare il framework per scrivere applicazioni J2ME

SYSML

EVOLUZIONI DEL LINGUAGGIO UML
La nuova specifica OMG orientata all'analisi dei sistemi complessi

PROGRAMMAZIONE 2.0

GOOGLE CONNECT
In 20 minuti il tuo sito web ospita tutte le funzionalità di un social network

YAHOO! USER INTERFACE
Le tabelle HTML con le funzionalità di editing tipiche di un foglio Excel

GOOGLE MAPPA LE INFO DEI TUOI DB
Ti mostriamo come rendere più immediato e divertente l'accesso ai dati

GOOGLE NATIVE CLIENT

FAR GIRARE IL CODICE X86 NEL BROWSER

Big G stupisce ancora. Ora i tuoi programmi si avviano in una pagina Web, così come delle normali applicazioni! Si apre la sfida ad Adobe Flash, Java e Microsoft Silverlight

SOLUZIONI: PROGRAMMARE UNA NUOVA REALTÀ CON IL RAY TRACING: LE TRASPARENZE E GLI EFFETTI CHE IMITANO I RIFLESSI DELL'ACQUA. TUTTE LE TECNICHE PER OTTENERE IMMAGINI FOTOREALISTICHE

EDIZIONI MASTER
www.edmaster.it

9 771128 594009 90136

CON CUDA IL TUO CODICE METTE LE ALI

CON IL PROGETTO CUDA DI NVIDIA, È POSSIBILE SCRIVERE APPLICAZIONI IN GRADO DI SFRUTTARE L'ENORME POTENZA DI CALCOLO DELLE MODERNE SCHEDE GRAFICHE. POTREMMO COSÌ REALIZZARE SOFTWARE CAPACE DI EFFETTUARE COMPLESSI CALCOLI IN POCHI SECONDI

L'acronimo GPGPU (*General Purpose computation on GPUs*) è conosciuto da alcuni anni grazie a progetti come *Folding@Home*, che utilizza BrookGPU, quando NVIDIA non aveva ancora consentito l'accesso alle proprie schede video. Indica quella tecnologia utilizzata per accedere alla potenza di calcolo delle GPU per effettuare operazioni su dati matematici/scientifici. Con il passare del tempo tale opportunità si è rilevata alquanto interessante, e quando si è iniziato a veder crescere un certo interesse, soprattutto da parte di organismi universitari e centri di ricerca, NVIDIA ha finalmente realizzato CUDA, nato pubblicamente nel 2006. Acronimo di *Computer Unified Device Architecture*, disponibile ora alla versione 2.1 e scaricabile gratuitamente dal sito http://www.nvidia.com/object/cuda_home.html e altresì disponibile nel supporto allegato alla rivista. CUDA rappresenta una delle soluzioni più avanzate, e soprattutto semplici, per sviluppare applicativi che richiedono l'analisi e/o la trasformazione di grandi quantità di dati in parallelo, fornendo un notevole risparmio di tempi di calcolo. La definizione più adatta per CUDA è quella di modello di calcolo parallelo e di ambiente di programmazione, il cui scopo è quello, a detta della stessa casa sviluppatrice, di facilitare l'esecuzione di calcoli matematici utilizzando linguaggi standard (per ora C e C++, FORTRAN a breve, pyCUDA come Wrapper Python, JCublas per Java e CUDA.NET per ambiente .NET). Nella pratica si tratta di una serie di librerie C che consentono di compilare applicativi in grado di interfacciarsi "direttamente" con la GPU. Perché NVIDIA ha rilasciato questa piattaforma software? Essere divenuto praticamente il monopolista nel campo della produzione e vendita delle schede grafiche, dopo l'acquisizione da parte di AMD di ATI, NVIDIA è riuscita, fino a oggi, a immettere sul mercato oltre 60 milioni di schede grafiche CUDA-enabled. Probabilmente è sembrato il momento opportuno perché tali periferiche oltrepassassero quel "recinto" di supporto grafico/ludico, e diventassero entità "parallele" alla CPU (veri e propri concorrenti in questo contesto). In un periodo in cui Intel sta valutando l'inserimento di schede video direttamente nei propri processori, questa decisione della società di Santa Clara sembra una mossa molto ponderata allo scopo di aumentare ulteriormente la fidelizzazione dei clienti. Grazie alla spinta fornita dai nuovi videogiochi, sempre più esigenti in termini di prestazioni, allo scopo di ottenere esperienze visive più verosimiglianti, in cui shader, luci dinamiche, particelle, trasparenze, alpha e altre tecniche sono praticamente onnipresenti, l'evoluzione tecnologica che le schede video hanno subito le ha rese dei dispositivi multithread, multicore, altamente parallelizzati e performanti. Schede corredate di memorie capienti e veloci. Le CPU, al contrario, sono state realizzate per effettuare un insieme eterogeneo di operazioni e la loro infrastruttura si è evoluta nel tempo proprio per adattarsi a tali necessità, mantenendo però una certa staticità (e un incremento di complessità che le ha rese sempre meno prestanti) allo scopo di evitare incompatibilità con software realizzati in precedenza. Per tale motivo sono stati aggiunti set di istruzioni, quali *MMX*, *3dNOW!*, *SSE1* e *2*, che, più di un'evoluzione del processore, trattasi di estensioni. L'ultima "soluzione", nata per aumentare le prestazioni complessive, è stata quello di inserire un certo numero di core all'interno della CPU fisica; il contrario è avvenuto invece, per i produttori di schede video, i quali hanno avuto piena libertà nel modificare a proprio piacimento l'hardware dei prodotti, a patto di mantenere comune l'API per interfacciarsi con esse. Perché non utilizzare solo le CPU attualmente sul mercato, fornite di due o quattro core e relativamente economiche in confronto ad una scheda video di medie prestazioni CUDA-enabled? Una GPU della famiglia GT200 (GTX 280) riesce a effettuare quasi 1000 GFLOPS/secondo (*FLoating point Operations*/secondo), operazioni in virgola mobile, mentre un Intel Core2 Duo di circa 3GHz arriva a malapena a 25

Conoscenze richieste

C

Software

Compilatore C e librerie CUDA

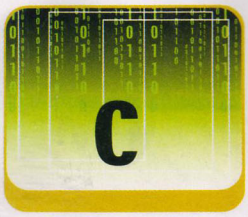
Impegno

Tempo di realizzazione

GFLOPS/secondo (il picco di 51,20GFLOPS/s è raggiunto dal Core2 Extreme processor e dal Core i7); se confrontiamo invece la velocità di trasferimento dei dati verso la memoria interna, una GPU famiglia G80 si avvicina al 100GB/s mentre una CPU circa 15GB/s. Alcuni modelli della serie GeForce 8, ad esempio l'8800 o il Tesla x870, possiedono 128 thread processor, chiamati anche *stream processor*, o anche *programmable pixel shader*. Ognuno di questi è in grado di supportare 96 thread in parallelo, per un totale massimo di 12288 thread eseguibili parallelamente! (il compilatore CUDA, per raggiungere tale numero di thread, dovrebbe assegnare massimo 10 registri per thread, ma in genere ne assegna una ventina, fino anche a 32, riducendo il numero effettivo a circa 4/6000, comunque un limite superiore che per molte CPU in commercio è praticamente irraggiungibile). Questi dati rendono ovvia la conclusione che, per avvicinarsi minimamente a tali prestazioni con una CPU, è necessario spendere decine di migliaia di euro (oltre al processore sono ovviamente necessari una motherboard appropriata, memoria RAM prestante, alimentatore sufficientemente potente etc) a fronte delle centinaia di euro necessarie per acquistare una scheda video di modeste prestazioni. Non solo, quindi, una riduzione di tempi di calcolo dell'ordine anche di cento volte, ma un notevole risparmio economico rendono questa soluzione come la migliore per risolvere problematiche di piccola-media entità (e grande se si utilizza il supporto di clustering di Rocks). Con una scheda video di qualche centinaia di euro si è in grado di ottenere una potenza di calcolo di 500 GFLOPS/s!

CUDA NELLA PRATICA

CUDA è utilizzato per la previsione della struttura delle proteine, per simulazioni meteo, analisi delle immagini medicali, astrofisica, reti neurali, ricerche oceanografiche, fisica delle particelle, chimica quantistica, astronomia, acustica, simulazioni elettromagnetiche e in decine di altri settori. Quanto sono migliorati quegli applicativi che hanno utilizzato CUDA? In alcuni test si è confrontato il tempo necessario per effettuare codifiche video utilizzando solo la CPU e si è notato una riduzione media da alcune ore a decine di minuti! Test su Matlab hanno rilevato miglioramenti di 17 volte. Testando CUDA con l'algoritmo quicksort si è registrata una velocità di esecuzione fino a 10 volte maggiore, operazioni di ray tracing sono state accelerate fino a 16 volte, calcoli di fluidodinamica fino a 17 volte, simulazioni fisiche di deformazioni di modelli tridimensionali fino a 20 volte, calcoli sulle proprietà delle molecole circa quattro volte, sistemi GIS (*Geographic Information System*) circa 36 volte, Mathematica, software realizzato dalla Wolfram, dichiara possibili miglioramenti da 10 a 100 volte nei calcoli e nelle simulazioni. Dai dati forniti dalle varie società, i miglioramenti sono comunque tali da giustificare l'utilizzo di tale tecnologia a fronte di una spesa molto ridotta. Nel campo medicale poniamo anche l'attenzione sul progetto FASTRA (*http://fastra.ua.ac.be*), realizzato dall'Università di Antwerp in Belgio. I ricercatori hanno realizzato un solo computer con una spesa modesta, sotto i 4.000,00€, utilizzando quattro GPU NVIDIA 9800GX2, costituite ognuna da un doppio processore, una CPU AMD Phenom e 8 GB di RAM, per effettuare in qualche ora ricostruzioni tomografiche che in campo medicale vengono effettuate con macchinari i cui costi sono quasi sempre proibitivi per molti istituti. I tempi necessari per effettuare tali calcoli con una singola CPU sarebbero dell'ordine di settimane, con CUDA si è riusciti ad arrivare ad alcune ore! Se l'incremento di prestazioni che hanno rilevato utilizzando una sola GPU CUDA-enabled era equivalente a circa 40 moderne CPU, utilizzandone quattro, inoltre dual core, ha permesso di ottenere l'equivalente di oltre 300 CPU core. La soluzione alternativa era quella di un cluster di computer, ma in questo caso si sarebbero presentati problemi di costi hardware, alimentazione, spazio, gestione: tutte spese che in questo modo non si sono presentate. Per chi volesse realizzare una configurazione simile ,basterà recarsi nella sezione *Specs & Benchamarks* per avere una lista dei componenti hardware installati. Ma oltre all'ambiente accademico/scientifico chi sta utilizzando CUDA? Alcuni cracker hanno trovato in CUDA la soluzione per effettuare ricerche di password utilizzando dizionari, oppure metodologie di bruteforce: in alcuni casi si è riusciti a trovare le chiavi di connessioni WPA in tempi ragionevoli, in altri con circa 5 secondi si è riusciti a trovare password criptate con MD5!!! La Cyberlink ha dichiarato che il suo software *PowerDirector* sarà accelerato in tal modo per l'elaborazione di video in alta definizione. Nero (conosciuta precedentemente come Ahead) in questo anno pare sia intenzionata a fornire tale supporto in una futura versione del suo software; TMPGEnc, software di encoding della Pegasys, utilizza già CUDA e ha rilevato un miglioramento di circa quattro volte; Badaboom Media Converter dichiara un miglioramento delle prestazioni di circa 20 volte, mentre la OptiTex, sviluppatrice di un software per la realizzazione e simulazione di tessuti, sta convertendo tutti i propri algoritmi in



OLTRE CUDA 2.0

NVIDIA prevede di estendere il supporto CUDA anche ad altri linguaggi, come C++ e Fortran. Saranno anche estese le funzionalità di debugging, oltre a garantire anche una migliore gestione dei profili.

una versione per GPU CUDA-enabled. Il mercato inizia quindi a muoversi verso questa tecnologia merito della sua facilità di sviluppo, potenza ed economia.

INTEGRAZIONE CON ALTRI SOFTWARE

Per facilitarne l'utilizzo da parte di organismi scientifici è stata sviluppata una plugin di integrazione con MathWorks MATLAB, liberamente scaricabile dal sito ufficiale di CUDA. È stata inoltre realizzata una plug-in per Adobe Photoshop, molto utile per realizzare nuovi filtri, allo scopo di ridurre i tempi di calcolo su immagini molto "pesanti", come quelle generate da dorsi digitali di medio/grande formato, o ottenute da scansioni in alta definizione, oppure anche per realizzare compositing di immagini per la realizzazione di HDR (immagini *High Dynamic Range*). È inoltre possibile creare un sistema di clustering utilizzando Rocks cluster, una distro Linux attualmente basata su CentOS. Queste due plugin, e l'integrazione con un sistema di clustering, forniscono in ambito scientifico tutto il necessario per consentire una consistente riduzione dei tempi, ma anche di risorse.

NOTA

LA CONCORRENTE FIRESTREAM

ATI Stream è la tecnologia di casa AMD che sfida NVIDIA CUDA. Per farlo AMD decide di entrare prepotentemente in questa fascia di mercato, proponendo la nuova soluzione FireStream 9270. Questa scheda integra un core da 750MHz, 2GB di memoria GDDR5 da 900MHz e una potenza computazionale di ben 1.2 teraflop.

SCHEDE VIDEO COMPATIBILI

L'elenco delle schede prodotte da NVIDIA che permettono di utilizzare CUDA è molto esteso, e per tale motivo ne stilo una rapida lista:

- NVIDIA GeForce serie 8 (ex. 8600, 8800), 9 (ex. 9600, 9800), 200 (ex. 280, 260) con minimo 256MB di RAM dedicata;
- NVIDIA Tesla (ex. 860, 1060, 1070);
- (NVIDIA Quadro (ex. 370, 2100, 3700, 4700).

Per un elenco completo recatevi all'URL www.nvidia.com/object/cuda_learn_products.html. Non sarà possibile utilizzare calcoli in virgola mobile se la scheda video non li supporta. Vorrei rassicurare i possessori dei modelli installati in portatili di fascia medio/alta, perché molti sono supportati, come le schede 8600M GT/GS (la prima è stata utilizzata proprio per redigere questo articolo). È comunque possibile sviluppare e testare i propri applicativi senza possedere uno dei sopracitati modelli, quindi senza un supporto hardware, adoperando la modalità di emulazione (disponibile direttamente nel menu di compilazione dei progetti in Visual C++), ovviamente non otterrete alcun vantaggio computazionale utilizzandoli in questo modo.

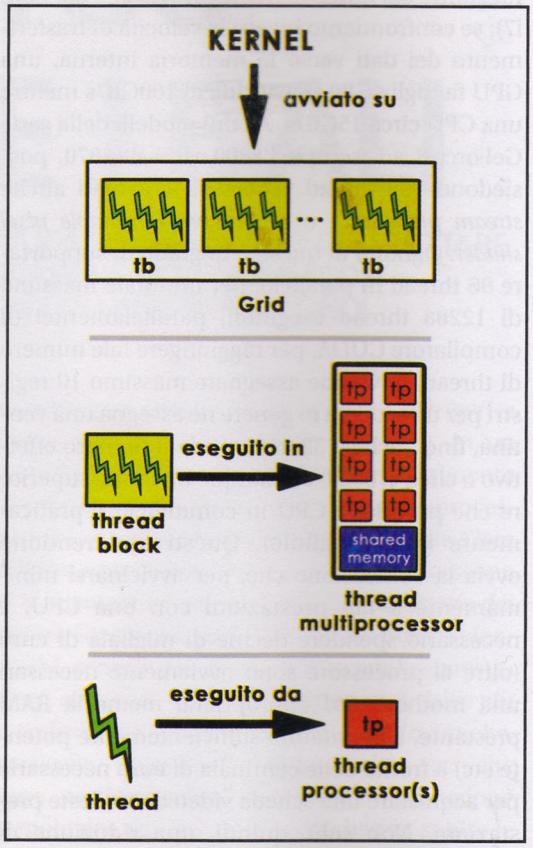


Fig. 1: La struttura di una GPU CUDA-enabled

CONCETTI FONDAMENTALI

Le parole chiave di questa tecnologia sono le seguenti:

- kernel;
- thread group;
- thread processor;
- memoria locale dei thread;
- memoria on-chip (shared memory) di un blocco;
- memoria on-board;

Il termine *Kernel* in questo contesto indica una parte di codice, una funzione, che viene eseguita all'interno della scheda video; ogni kernel/funzione viene eseguito sequenzialmente (*funzione1...funzioneN*) in un grid di blocchi di thread, chiamati *thread block* (identificati con coordinate bidimensionali *x, y*), come *n* thread software; ogni thread block viene eseguito in un *thread multiprocessor* (più thread block possono essere eseguiti su un solo multiprocessor, il limite è imposto dall'hardware), i quali contengono a loro volta x *thread processor*, forniti di registri e una propria memoria *locale* (non bufferizzata,

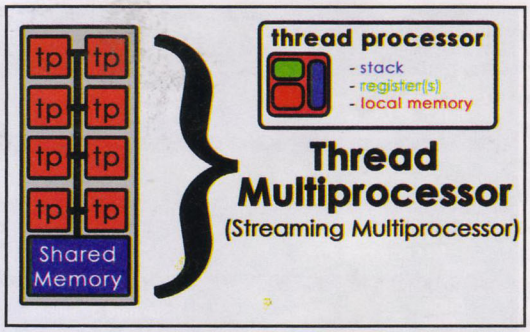
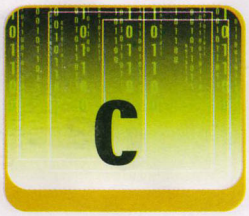


Fig. 2: La composizione interna di un thread multiprocessor

più lenta di quella on-chip, e il cui tempo di vita si esaurisce con quello del thread stesso). Quanti thread block possono essere inseriti in un thread multiprocessor dipende da quanti registri vengono assegnati ad ogni thread e quanta memoria *on-chip* (shared) è assegnata ad ogni blocco.
I thread processor sono quindi le unità atomiche della nostra scheda: il codice del kernel viene infine eseguito in questi ultimi. Un sistema di "scatole cinesi" relativamente semplice. La scheda video G80, ad esempio, ha 128 thread processor, suddivisi in 16 thread multiprocessor, ognuno contenente 8 thread processor. Nel caso di GPU che supportano la doppia precisione, questa unità è presente in ogni multiprocessor. Ogni thread è corredato di uno stack, un program counter, una propria memoria locale e una serie di registri. I thread utilizzati da CUDA sono lightweight, ma non devono essere sottovalutati, per tale motivo sorge immediata la problematica dei *Deadlock*. NVIDIA dichiara che è praticamente impossibile trovarsi in questa situazione se si programma seguendo la documentazione fornita, quindi utilizzando chiamate a funzioni documentate dell'API (anche se è sempre consigliabile prendere in considerazione tale possibilità). Ogni scheda video ha una determinata memoria fisica, on-board (è necessario averne dedicati almeno 256MB), che potremo utilizzare senza problemi, ma grazie a NVIDIA abbiamo anche accesso a quei blocchi di memoria condivisa da gruppi di thread processor chiamati *on-chip memory/shared memory*. Tale memoria consente di condividere informazioni, come strutture dati e variabili (ecco il vantaggio rispetto alle soluzioni GPGPU, che non hanno modo di arrivare a tali blocchi di memoria); le *on-chip memory* hanno inoltre il pregio di ridurre la necessità di accedere alla memoria *on-board*, di alcuni ordini di grandezza più lenta (come avviene con le cache di n-livello nelle CPU).
Da tutte queste informazioni si delinea la necessità di cambiare, per molti radicalmente, la propria ottica di sviluppo software, per arrivare ad una visione dove si deve considerare anche il modo di suddividere e analizzare i flussi di informazioni. Per ottimizzare l'utilizzo della GPU è necessario quindi bilanciare il numero di thread per blocco, la memoria richiesta da un singolo blocco (e sottratta a quella condivisa fornita dal multiprocessor), e il numero di registri utilizzati da ogni singolo thread; per effettuare tale misurazione è disponibile un foglio di calcolo, fornito come allegato all'articolo (e anche presente nella cartella *tools* quando si installa l'SDK), che, opportunamente compilato, restituirà una serie di grafici e indici numerici allo scopo di mostrare un feedback immediato sulle prestazioni che otterrete, il tutto senza dover effettuare preventivamente test sul vostro codice. I dati da inserire sono soltanto quattro e consistono in:

- Compute Capability della vostra GPU (spiegata in un paragrafo successivo);
- Numero di Thread per Blocco;
- Registri da associare a ogni Thread;
- Memoria condivisa (byte) utilizzata per blocco di thread;

I thread devono inoltre essere eseguiti in gruppi di almeno 32 per ottimizzare le prestazioni.
Oltre a una ottimizzazione sul numero di thread e all'attribuzione delle risorse, è necessario prendere in considerazione che in questo contesto il vero collo di bottiglia è dovuto soprattutto al trasferimento delle informazioni tra GPU e CPU, ma si evidenzia anche quando si accede alla memoria *on-board* quando le informazioni non sono disponibili in quella *on-chip*. Quando un thread necessiterà di utilizzare informazioni esterne al proprio thread multiprocessor, quindi alla memoria *on-board*, verrà messo in attesa, il suo posto sarà occupato immediatamente da un altro thread precedentemente in attesa di esecuzione. Quando i dati richiesti diventeranno disponibili, passerà a un'altra lista d'attesa, dopo la quale ritornerà in esecuzione. La scelta di quale thread eseguire avviene tramite l'algoritmo *round-robin*, fornendo, in maniera equa, tempo di calcolo a ogni thread. Per chi si chieda se sarà necessario programmare codice per gestire questa situazione di concorrenza la risposta è negativa: viene tutto gestito automaticamente dalla scheda video.
Un esempio è l'utilizzo di CUDA negli antivirus: un tipico software di questa famiglia confronta migliaia di signature/firme/impronte digitali di altrettanti virus, in maniera spesso sequenziale, con quella di ogni file che sta analizzando; utilizzando CUDA si potrebbe assegnare ad ogni thread una signature di un virus diverso e utilizzare la memoria condivisa del thread multiprocessor (*on-chip*) per memorizzare quella del file analizzato,



SOFTWARE CUDA ENABLED

Sono già diverse le applicazioni che sfruttano o sfrutteranno a breve la potenza di CUDA o di tecnologie similari. Tra queste: la suite ADOBE CS, CyberLink PowerDirector 7, ArcSoft TotalMedia Theater e Microsoft Windows Vista, Expression Encoder, Office PowerPoint 2007, Silverlight etc.

consentendo un'analisi contemporanea di migliaia di signature! In questo modo si utilizzerebbe la CPU solo per un ridotto numero di operazioni, demandando tutto il carico di lavoro alla GPU. La società Kaspersky ha realizzato una versione del proprio antivirus utilizzando la tecnologia concorrente realizzata da ATI, Stream, chiamata Kaspersky SafeStream anti-virus, e ha dichiarato miglioramenti di venti volte, con un utilizzo della CPU, in fase di scansione, del solo 2%!!!

NEI PENSIERI DI INTEL

Intel starebbe pensando di introdurre presto nuovi processori con GPU integrata. Il progetto, nome in codice Larrabee, prevederebbe l'impiego di un engine a virgola mobile, grazie al quale si dovrebbe superare in modo netto la potenza di elaborazione di un Teraflop. Intel così potrà sfidare NVIDIA Tesla e AMD Firestream.

COMPUTE CAPABILITY

Con questo termine si indica la famiglia di appartenenza di ogni singola scheda video, e permette di identificare le funzionalità, e i limiti, che CUDA ha modo di utilizzare. Attualmente sono quattro: 1.0, 1.1. 1.2 e 1.3. Sul sito NVIDIA troverete materiale informativo che vi indicherà la categoria della vostra scheda video:

Compute Capability 1.0:

- Ogni blocco non può avere più di 512 thread;
- Ogni blocco può avere dimensioni massime *x=512, y=512, z=64*;
- Un grid di thread block può avere dimensione massima per "lato" di 65535;
- La dimensione del warp è di 32 thread;
- Ogni multiprocessor ha 8192 registri;
- Ogni multiprocessor ha al massimo 16 KB di memoria condivisa, suddivisa in 16 banchi;
- Ogni multiprocessor può eseguire al massimo 8 thread block in parallelo, 24 warp in parallelo e 768 thread in parallelo;
- [...]

Compute Capability 1.1:

- Supporto per operazioni atomiche su word di 32bit nella memoria globale (on-board);

Compute Capability 1.2:

- Supporto per operazioni atomiche su word di 32bit nella memoria condivisa (on-chip), e di 64 bit su quella globale (on-board);
- Ogni multiprocessor ha 16384 registri, può eseguire al massimo 32 warp in parallelo e massimo 1024 thread in parallelo;

Compute Capability 1.3:

- Supporto per numeri in virgola mobile a doppia precisione;

IL PROCESSO DI COMPILAZIONE

Scrivere un applicativo CUDA necessita la creazione di soli due file, uno indirizzato per essere eseguito sulla GPU e l'altro sulla CPU, poiché questa operazione viene effettuata automaticamente dal compilatore; il vostro codice sarà suddiviso e compilato in due parti: la prima, destinata alla CPU, l'altra (utilizzando il compilatore Open64) nel formato PTX (*Parallel Thread eXecution*), indipendente dall'hardware della scheda video. Questo processo ha alcuni vantaggi, consente di scrivere file con codice eterogeneo, utilizza compilatori diversi ottimizzati per le rispettive piattaforme e non richiede la ricompilazione della parte dedicata alla GPU, poiché il formato PTX, come detto, è indipendente dalla specifica scheda.

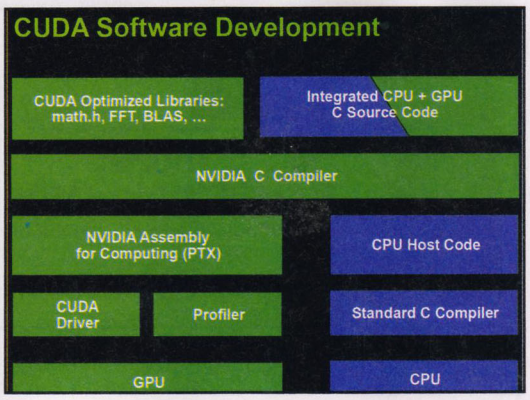


Fig. 3: La struttura della tecnologia CUDA

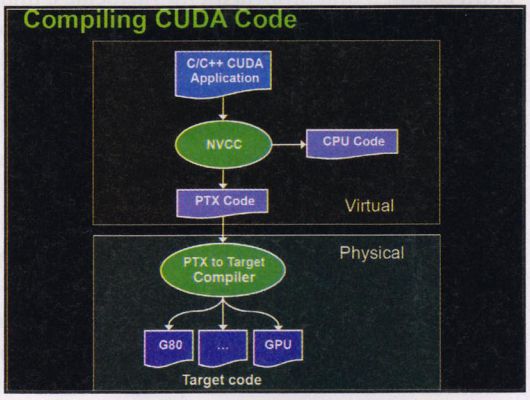


Fig. 4: Come avviene il processo di compilazione in due "direzioni"

INSTALLAZIONE E CONFIGURAZIONE

CUDA è disponibile sia per ambiente Windows XP/VISTA (32/64 bit), associato a Visual Studio (incluse le versioni gratuite Express) che per Mac OS e distro Linux (32/64 bit) come Fedora, Ubuntu, Red Hat, SUSE. In questo articolo ci concentreremo solo della versione Windows; sul sito ufficiale è ovviamente disponibile tutta la documentazione necessaria per utilizzarlo sugli

altri due sistemi operativi. Come prima operazione è necessario verificare che la propria scheda video sia supportata (fate riferimento al link presente nel paragrafo precedente). Recatevi su *http://www.nvidia.com/object/cuda_get.html* e scaricate una versione dei driver ForceWare uguale, o superiore, alla 177.35; in caso abbiate una versione relativamente aggiornata consiglio di utilizzate quella fornita direttamente nella pagina di download di CUDA, poiché è stata preventivamente testata per essere utilizzata con tale framework; per i modelli Mobile conviene, invece, utilizzare i driver reperibili dal sito NVIDIA, una versione meno recente, poiché quelli forniti nella pagina di CUDA non sono compatibili (v180.60). Scaricate sempre dalla stessa pagina i due package disponibili, di cui uno opzionale: CUDA Toolkit, obbligatorio per effettuare la compilazione dei propri software, e CUDA SDK, contenente applicativi che dimostrano alcuni utilizzi "tipici" in campo prettamente scientifico-matematico. Il Toolkit comprende i seguenti file e documenti:

- il compilatore C *NVCC*;
- le librerie CUDA FFT (Fast Fourier Transform) e BLAS (Basic Linear Algebra Suprograms);
- un Profiler;
- un debugger gdb per la GPU;
- CUDA runtime driver;
- documentazione.

CUDA FFT permette di applicare la Trasformata di Fourier senza dover realizzare una propria implementazione: consente di applicare tale metodo in una, due o tre dimensioni, l'esecuzione di trasformazioni multiple a singola dimensione in parallelo, trasformazioni fino a 8 milioni di elementi in singola dimensione, fino a 16384 per la doppia e tripla dimensione.
CUDA BLAS consente di effettuare operazioni su vettori e matrici, è una tecnologia standard (definita come un'API) de facto, implementata da molti vendor: IBM, INTEL, HP e NEC per citarne alcuni. È suddivisa in tre livelli: il primo effettua operazioni scalari e su vettori, il secondo tra vettori e matrici, il terzo tra matrici.

L'SDK fornisce alcuni demo, disponibili sia come sorgenti sia eseguibili:

- Parallel Bitonic Sort (algoritmo di ordinamento specifico per calcolo parallelo);
- Moltiplicazioni e trasposizione di matrici;
- Operazioni di convoluzione su immagini (blur, edge detection, rimozione rumore etc);
- Post Processing su OpenGL e Direct3D;
- Esempi di utilizzo di BLAS e FFT;
- Integrazione tra CPU-GPU tramite C e C++;
- Binomial Option Pricing, Black-Scholes Option Pricing, Monte-Carlo Option Pricing (utilizzate in campo economico/statistico per misurare il valore di una stock option);
- Parallel Mersenne Twister (generazione di numeri pseudocasuali);
- Calcolo di istogrammi e rimozione rumore da immagini;
- Sobel Edge Detection Filter (utilizzato per identificare i margini degli oggetti presenti in un'immagine);

VISUAL C++ 2008 EXPRESS E CUDA 64BIT

Per gli utenti Visual C++ 2008 Express. Il software Microsoft non include un compilatore x64, quindi non riuscirete neppure ad avviare il processo di build dei progetti, inoltre, se proverete a compilare con target 32bit non sarà possibile a causa della mancanza di alcune librerie, non fornite con CUDA 64bit; dovrete scaricare e utilizzare manualmente il compilatore a 64bit fornito con il Platform SDK Microsoft, utilizzare una versione di Visual Studio che includa tale supporto, oppure capitolare per la versione a 32bit dell'sdk/toolkit. In ultima alternativa è possibile cambiare piattaforma (utilizzando ad esempio distro Linux a 64bit) .

Se per eseguire i demo presenti nell'SDK basta selezionare la relativa voce nell'interfaccia grafica installata, per compilarli, invece, sarà necessario effettuare una serie di operazioni per configurare Visual C++; per semplificare le operazioni abbiamo allegato il CUDA VS Wizard, plugin (funzionante anche per la versione Express) che permette di creare un progetto per il software NVIDIA con una serie di semplici clic. Nel caso in cui, in fase di compilazione, non venisse trovata la variabile *NVSDKCUDA_ROOT* dovrete crearla come variabile di sistema, oppure sostituirla con la path dell'SDK, nel nostro caso è *C:\ProgramData\NVIDIACorporation\NVIDIA CUDASDK*. Se quando eseguirete in debug/release mode, il sistema non troverà i file cutil32.dll e cutil32D.dll, copiate quest'ultimi nella cartella in cui vengono creati gli eseguibili (oppure impostateli in *Configuration Properties Linker Input Additional Dependencies= cudart.lib cutil32.lib*).

NEL DETTAGLIO...

L'API è una estensione del linguaggio C ed è suddivisa in due componenti:

- una serie di estensioni del linguaggio, per identificare quali parti devono essere eseguite

sulla scheda video;
- una libreria a runtime suddivisa a sua volta in tre parti:
  1 - una il cui scopo è fornire il controllo e l'accesso alle schede video da parte della CPU;
  2 - un'altra di funzioni eseguibili lato GPU;
  3 - un'ultima, comune, per creare e utilizzare variabili e vettori utilizzati in entrambi gli ambienti (GPU e CPU).

Programmare con CUDA consiste nell'inviare dati alla GPU e richiamare opportune funzioni su tali dati, effettuare altre elaborazioni in parallelo, utilizzando la CPU e, infine, reperire le informazioni elaborate dalla GPU, quando disponibili. Il codice eseguito sulla CPU è responsabile dell'invio, della ricezione e dello spostamento delle informazioni verso/dalla/nella memoria della scheda video (quella globale *on-board*). I kernel sono quindi funzioni C, che automaticamente copiano nella scheda video gli argomenti passati, ma che hanno alcune limitazioni:

- non possono accedere alla memoria della CPU;
- devono restituire *void*;
- non possono avere un numero variabile di parametri;
- non possono essere ricorsive;
- non possono utilizzare variabili statiche;

Se tutti i kernel devono restituire *void*, com'è possibile ottenere un feedback se avviene qualche errore nel codice?
Bisogna invocare la funzione *cudaError_t cudaGetLastError(void)* per identificare il codice d'errore, mentre *char* cudaGetErrorString(cudaError_t code)* per il messaggio d'errore, basterà quindi richiamare:

```
printf("%s\n", cudaGetErrorString(
                                cudaGetLastError() ) );
```

per verificare se ci sono stati problemi nell'ultima esecuzione.
Come è possibile decidere e identificare quali funzioni/kernel verranno eseguite sulla CPU e quali sulla GPU? Per questo esistono i seguenti comandi:

- **__global__** : la funzione sarà richiamata sulla CPU ed eseguita sulla scheda video; obbligatoriamente, come suddetto, deve restituire void;
- **__device__** : la funzione sarà richiamata, ed eseguita, sulla scheda video;
- **__host__** : la funzione sarà richiamata, ed eseguita, sulla CPU (il comportamento standard);

Per le variabili:

- **__constant__** : allocata come costante nella GPU, è accessibile dai thread nel grid e dall'host; oltre ai registri, rappresenta la posizione in cui si ottengono maggiori prestazioni quando si richiede una "variabile";
- **__device__** : allocata nella memoria *on-board*, è accessibile dai thread nel grid e dall'host;
- **__shared__** : allocata nella memoria condivisa *on-chip*, termina con la fine dell'esecuzione del block, è accessibile solo ai thread del block;

Tali qualificatori possono essere utilizzati insieme in alcune configurazioni, ma hanno comunque precisi vincoli e limitazioni; in caso non venga utilizzato uno di questi qualificatori, la variabile verrà posizionata in un registro, e in caso sia di dimensioni non compatibili, verrà posizionata nella memoria locale.
Come si crea quindi un kernel che viene richiamato sulla CPU, ma eseguito sulla GPU? Prima di tutto si crea la funzione, che, ripetiamo, deve restituire *void*, poi si antepone *__global__*

```
__global__ void nomekernel(parametro1,..,prametro n)
{
   //codice funzione
}
```

Si deve poi richiamarla comunicando il numero di thread block e il numero di thread per ogni thread block che la andranno a eseguire:

```
nomekernel<<<dim3 grid, dim3
     threadsPerBlock>>>(parametro1,..,parametro n)
```

- **grid**: indica il numero di thread block da avviare; il prodotto *grid.x * grid.y* è quindi il numero complessivo di blocchi allocati per ogni grid;
- **threadsPerBlock**: indica il numero di thread per blocco; è ottenuto dal prodotto *threads PerBlock.x*threadsPerBlock.y *threadsPer Block.z*, che indicano il numero di thread in ognuna delle tre dimensioni;

L'omissione di uno dei campi *x, y, z* comporta l'assegnazione del valore di default unitario.

```
dim3 grid, block;

grid.x = 2; grid.y = 4;
```

SVILUPPARE CON INTEL

A differenza di CUDA, la nuova soluzione Intel dovrebbe consentire a qualunque sviluppatore di scrivere applicazioni così come avviene per un qualunque altro microprocessore. Non sarà quindi necessario imparare a utilizzare nuove API e apprendere nuovi concetti. Chiunque dovrebbe poter scrivere fin da subito applicazioni in grado di sfruttare l'enorme potenza di calcolo.

```
block.x = 8; block.y = 16; //block.z = 1 per default
nomekernel<<<grid,
            block>>>(parametro1,..,parametro n)
```

Eseguirà la funzione su 8 (2x4) blocchi, per ogni blocco ci saranno 128 (8x16x1) thread, poichè block.z assume il valore di default 1, mentre:

```
grid.x = 1; //grid.y=1 per default
block.x = 8;block.y=8; //block.z=1 per default

nomekernel<<<grid,
            block>>>(parametro1,..,parametro n)
```

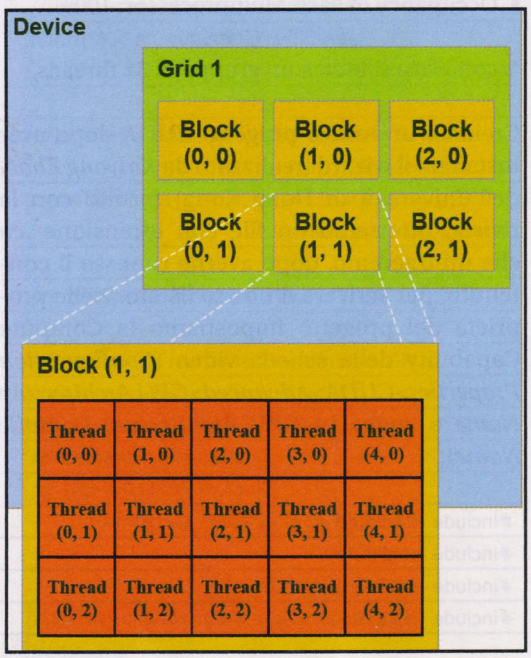


Fig. 5: Esempio di coordinate di gruppi all'interno di un grid

eseguirà la funzione su un blocco (1x1), poiché *grid.y* ha come valore di default 1, e per tale blocco saranno utilizzati 64 (8x8x1) thread poiché *block.z* assume il valore di default 1. Alcune variabili sono automaticamente rese disponibili se si definisce una funzione *__global__ o __device__*:

- **dim3 gridDim**: dimensione del grid (numero di blocchi per singola dimensione);
- **dim3 blockDim**: numero dei threads per blocco (per singola dimensione):
- **dim3 blockIdx**: indice del blocco nel grid ;
- **dim3 threadIdx**: indice del thread nel blocco;

Ogni kernel viene eseguito in maniera asincrona, quindi si avrà immediatamente il controllo della CPU dopo aver chiamato tale funzione, bisogna tenere in considerazione che l'ultimo kernel inviato alla scheda video dovrà attendere il termine dei precedenti. La copia da/verso/nella memoria, utilizzando la funzione *cudaMemcpy()* è sincrona, quindi bisogna attendere il termine dell'operazione, e anche questa dovrà attendere il termine delle precedenti chiamate/kernel in esecuzione nella GPU. La funzione *cudaMemcpy* è fondamentale e permette di copiare dati da/verso/nella memoria della GPU; per copiare il contenuto di variabile presente nella CPU in una nella GPU è necessario prima creare una variabile nella scheda video tramite il metodo *cudaMalloc*: in questo esempio allochiamo 100 locazioni contigue per contenere altrettanti float.

```
float *a_d;
int NBytes = 100*sizeof(float);
cudaMalloc((void **) &a_d, nBytes);
```

In questo modo abbiamo allocato tale variabile all'interno della memoria *on-board*, più lenta, ma di maggiori dimensioni, accessibile da tutti i thread e il cui tempo di vita è quello del nostro applicativo. Nel caso, invece, avessimo voluto allocarla in quella di minori dimensioni, *on-chip*, dovevamo anteporre *__shared__* al tipo della variabile; in questo caso il tempo di vita sarebbe stato limitato alla durata del blocco di thread.
Variabili scalari e vettori "predefiniti" *([u]char[1..4], [u]short[1..4], [u]int[1..4], [u]long[1..4], float[1..4], double[1..2])* vengono invece memorizzati nei registri, a meno che non siano di dimensioni tali da necessitare un'allocazione nella memoria locale del thread (che ricordo più lenta di quella condivisa *on-chip*);
Ora è necessario copiare il contenuto della variabile nella memoria della GPU, sapendo che la prima variabile accettata dalla funzione è sempre quella di destinazione:

```
cudaMemcpy(variabile_GPU, variabile_CPU,
                dimensioniByte, cudaMemcpyHostToDevice);
```

Per assegnare il valore di una variabile presente nella GPU a un'altra sempre nella stessa periferica si dovrà utilizzare:

```
cudaMemcpy(variabile_GPU, variabile_GPU,
                                    dimensioniByte,
cudaMemcpyDeviceToDevice);
```

Infine, per assegnare il valore di una variabile presente nella GPU a una presente nella CPU, si dovrà utilizzare:


```
cudaMemcpy(variabile_CPU, variabile_GPU,
        dimensioniByte, cudaMemcpyDeviceToHost);
```

Una tipica esecuzione di un applicativo consiste quindi nel:

1. allocare tramite *cudaMalloc* le variabili necessarie nella GPU;
2. copiare dati verso la GPU tramite *cudaMemcpy*;
3. avviare uno o più kernel/funzioni per elaborare tali informazioni sulla GPU;
4. (opzionalmente) effettuare operazioni lato CPU nel mentre;
5. riottenere i dati elaborati tramite *cudaMemcpy* e utilizzarli/mostrarli/memorizzarli lato CPU.
6. liberare la memoria allocata nella GPU utilizzando *cudaFree*;

IL PRIMO APPLICATIVO

Un semplice software che modifica ogni elemento di un array di float (allocato dinamicamente e con dimensione di 1.280.000 elementi), moltiplicando ogni elemento dell'array per 10, eseguito sulla CPU sarà codificabile come segue:

```
#include <stdio.h>
#include <stdlib.h>

void increment_cpu(float *a, int N)
{
        for (int i = 0; i<N; i++)
        a[i] *= 10;
}

void main(){
        [...]
        increment_cpu(a, 1280000);
}
```

Una versione CUDA utilizzerà invece le coordinate di ogni thread per effettuare operazioni solo su una certa posizione nell'array, in questo modo un solo thread è associato univocamente ad essa. Il numero di grid (numero di blocchi da utilizzare) è calcolato suddividendo le 1.280.000 locazioni dell'array in gruppi di 256, in tal modo avremo 5000 blocchi da 256 thread, visti i limiti delle GPU con compute capability 1.1 il numero massimo di thread (768) viene raggiunto utilizzando tre blocchi (256*3) e consente di avere un utilizzo ottimale (100%) di ogni multiprocessor; ogni thread nel suo gruppo avrà un *threadIdx.x* da 0 a 255; questi valori sono stati ottenuti inserendo le quattro informazioni necessarie nel foglio di calcolo per verificare l'effettivo utilizzo della GPU:

- Compute Capability: 1.1
- Threads Per Block: 256
- Registers Per Thread: 0
- Shared Memory Per Block (bytes): 0

Il risultato di tale inserimento è il seguente:

- Active Threads per Multiprocessor: 768
- Active Thread Blocks per Multiprocessor: 3 (3x256=768)
- Active Warps* per Multiprocessor: 24
- Occupancy of each Multiprocessor: 100%

*: con Warp si indica un gruppo di 32 threads.

Creiamo un nuovo progetto CUDA dopo aver installato il wizard (realizzato da *Kaiyong Zhao*, dell'università di Hong Kong) fornito con la rivista, otterremo un file con estensione *.cu* che utilizzeremo, dopo averne rimosso il contenuto, per scrivere il nostro listato. Nelle proprietà del progetto impostiamo la Compute Capability della scheda video (*Configuration Properties>CUDA>Advanced>GPU Architecture Name* e anche in *GPU Architecture Compile Name*);

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cutil.h>

__global__ void inc_array(float *a, int N)
{
  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  if (idx<N) a[idx] *=10;
}

int main(void)
{
  float *a_h, *a_d;

  const int N = 1280000;
  size_t size = N * sizeof(float);

  //Allochiamo in "locale" la struttura dati
  a_h = (float *) malloc(size);

  //Attribuiamo come valore il proprio indice
  for (int i=0; i<N; i++) {
        a_h[i] = (float)i;
  }

  //Allochiamo una versione speculare nella GPU
  cudaMalloc((void **) &a_d, size);
```

```
//Copiamo l'array nella GPU
cudaMemcpy(a_d, a_h, size,
                        cudaMemcpyHostToDevice);
int block_size = 256;
int n_blocks = N/block_size + (N%block_size == 0
                                        ? 0:1);

printf("Calculating ^2 of %d elements
                                    array...\n",N);

//Avviamo il kernel
inc_array <<< n_blocks, block_size >>> (a_d, N);

//Verifichiamo che non ci siano stati errori
printf("%s\n", cudaGetErrorString(
                                cudaGetLastError() ) );

//Copiamo il contenuto dell'array della GPU in
                                    quello della CPU
cudaMemcpy(a_h, a_d, sizeof(float)*N,
                        cudaMemcpyDeviceToHost);

printf("Operazione completata\n");

//Liberiamo la memoria allocata sulla GPU
cudaFree(a_d);

//Mostriamo il risultato
for (int i=0; i<N; i++) {
        printf("%d %f\n", i, a_h[i]);
}

//Liberiamo la memoria allocata sulla CPU
free(a_h);
}
```

L'indice *idx* dell'array viene calcolato per ogni thread utilizzando le proprie coordinate rispetto al proprio grid e blocco: *int idx = blockIdx.x * blockDim.x + threadIdx.x*; se il thread si trova nel 5° blocco e 6a posizione avremo: *(4*256)+5=1029*, andrà quindi ad analizzare l'indice in 1029a posizione.

ALTRE TECNOLOGIE

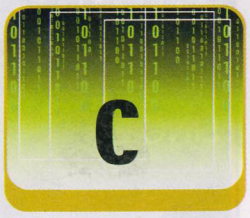
Penso sia chiaro che sviluppare utilizzando questa tecnologia, come tutte quelle di calcolo parallelo, non è un'attività semplice, ma i vantaggi che si possono ottenere sono tali da giustificare un maggiore impegno di programmazione: è un investimento che ripagherà in molti modi.

NVIDIA non è l'unica società che ha realizzato software per supportare il calcolo su GPU. L'ultima arrivata è *RapidMind* con il suo *Multicore Development Platform*. Consente di utilizzare sia NVIDIA, che ATI, che CELL (il "cuore" della Playstation 3), ha comunque alcune limitazioni, come il supporto per le sole cpu x86 e un prezzo non reso pubblico; PeakStream aveva realizzato delle librerie per le GPU ATI nel 2006, ma nel 2007 è stata acquisita da Google e se ne sono quindi perse le tracce; Intel ha rilasciato, attualmente in fase BETA gratuita (la versione finale sarà quindi a pagamento), Intel Parallel Studio, add-on di Visual Studio molto interessante. Arriviamo al concorrente diretto di NVIDIA nel settore delle schede video: ATI.

La tecnologia *Stream*, rilasciata dalla società ormai parte integrante di AMD, sembra molto interessante e dai loro report risulterebbe essere ancora più prestante, per alcune configurazioni, rispetto alle soluzioni del suo concorrente; l'SDK è liberamente scaricabile, non supporta come CUDA le funzioni ricorsive, e un numero variabile di parametri, ma ha un difetto che potrebbe essere la più probabile causa del suo ridotto utilizzo: il numero di GPU che lo supportano non sono tante quanto quelle che NVIDIA ha già piazzato sul mercato; resta comunque una valida alternativa da prendere in considerazione nel caso si dovesse costruire un sistema parallelo ad-hoc; il mercato dell'usato è ovviamente più dinamico sempre per NVIDIA (consentendo di realizzare sistemi CUDA-accelerated con ancora meno risorse economiche).

Un terzo concorrente, che però sembra più un pacificatore, sarà OpenCL (*Open Compute Library*), neo-fratello ideologico di OpenGL e OpenlAL, nato in seno Apple e disponibile con Mac Os X 10.6 (Snow Leopard), supportato da grandi gruppi come Intel, Motorola, IBM, AMD, Nokia, Motorola, NVIDIA compresa, e che vuole divenire un'API a basso livello standard, per realizzare soluzioni open di calcolo parallelo utilizzando CPU/GPU (incluso CELL): NVIDIA non ne ha alcun timore, anzi lo supporta, e infatti ha già implementato il supporto in CUDA 2.1 per Open CL versione 1.0. Il difetto imputato da molti a CUDA è quello di essere utilizzabile solo con le schede video della sua sviluppatrice, ma ha come pregio l'immediata disponibilità, il fatto che è gratuito e viene testato giornalmente in molti ambiti scientifici/professionali. Resta il fatto che tutte queste soluzioni gratuite possono aiutare decine, se non centinaia, di università e centri di ricerca nel mondo (ma soprattutto quelli del nostro paese, così bisognosi di risorse...). Buona programmazione.

Andrea Leganza



L'AUTORE

Laureato in Ingegneria Informatica, da oltre un decennio realizza soluzioni multimediali e software su diverse piattaforme e linguaggi. Certificato EUCIP Core, appassionato di fotografia, lingua giapponese, tiro con l'arco, è istruttore di nuoto FIN; attualmente è programmatore indipendente iPhone; è contattabile su neogene@tin.it o direttamente sul sito http://www.leganza.it