Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

# Accelerare L'Algebra Lineare con OpenCL

Tesi di Laurea in Architettura degli Elaboratori

Relatore:
Chiar.mo Prof.
Vittorio Ghini

Presentata da:
Mauro Belgiovine

Sessione III
Anno Accademico 2011/2012

*Alla mia famiglia,*
*a Chiara,*
*a mio fratello,*
*agli amici di ieri, di oggi e di domani*
*per la pazienza e il supporto…*

# Introduzione

L'evoluzione dei microprocessori e la loro crescente potenza di calcolo hanno accompagnato lo sviluppo di numerosissimi settori nel campo del'economia e della ricerca scientifica. Tra le varie discipline inerenti il calcolo legato alla computazione matematica, l'Algebra Lineare ricopre un ruolo molto importante nella ricerca: è possibile, ad esempio, formalizzare un certo fenomeno in un sistema lineare e studiarne gli sviluppi sottoponendolo a trasformazioni di varia natura, assimilabili alla risoluzione di questi sistemi.

Dal punto di vista computazionale, il calcolo algebrico è stato limitato negli anni dal'inadeguatezza dell'hardware, non ancora capace di eseguire efficacemente questo tipo di elaborazioni.

L'aumento dell'interesse per il calcolo parallelo, in particolare delle tecnologie legate al GPGPU e alla programmazione eterogenea, hanno portato alla creazione di sistemi di calcolo molto potenti e reattivi che, rispetto ai costosi e dispendiosi super-calcolatori, permettono di raggiungere ottime performance computazionali, mantenendo bassi i consumi energetici e gli investimenti in tecnologie specializzate nel calcolo massivo.

OpenCL è un framework dedicato alla programmazione parallela di sistemi eterogenei: essendo uno standard aperto e adottato da un numero sempre maggiore di produttori hardware, esso si propone come un efficiente libreria per la creazione di applicazioni altamente performanti e portabili.

Con il mio lavoro di tesi ho voluto evidenziare la possibilità di incrementare le performance per questo tipo di computazioni mediante l'uso di hardware specializzato nel calcolo parallelo, in particolare dell'uso di OpenCL

e del calcolo ibrido di questi algoritmi su CPU e GPU.

# Indice

**Introduzione** **i**

**1 Scenario** **1**

1.1 Calcolo Parallelo . . . . . . . . . . . . . . . . . . . . . . . . . 1

1.2 GPGPU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 2

1.3 Programmazione Eterogenea . . . . . . . . . . . . . . . . . . . 5

1.4 OpenCL . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 6

**2 Obiettivo** **11**

2.1 Algebra Lineare . . . . . . . . . . . . . . . . . . . . . . . . . . 11

2.2 MAGMA . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 12

2.3 Algoritmi a blocchi e limiti di scalabilità . . . . . . . . . . . . 13

2.4 Approccio per sistemi multicore paralleli . . . . . . . . . . . . 15

2.5 Differenze tra MAGMA e clMAGMA . . . . . . . . . . . . . . 17

2.6 Modifiche proposte per clMAGMA . . . . . . . . . . . . . . . 19

**3 Implementazione** **23**

3.1 Inizializzazione dinamica dell'ambiente . . . . . . . . . . . . . 23

3.1.1 Implementazione . . . . . . . . . . . . . . . . . . . . . 27

3.1.2 Sviluppi futuri . . . . . . . . . . . . . . . . . . . . . . . 34

3.2 Algoritmi multi-device . . . . . . . . . . . . . . . . . . . . . . 34

3.2.1 Implementazione . . . . . . . . . . . . . . . . . . . . . 38

3.2.2 Sviluppi futuri . . . . . . . . . . . . . . . . . . . . . . . 50

3.3 Kernel OpenCL cross-platform . . . . . . . . . . . . . . . . . . 51

3.3.1 Sviluppi futuri . . . . . . . . . . . . . . . . . . . . . . . . 51
3.4 Auto-bilanciamento del carico di lavoro . . . . . . . . . . . . . 51
3.4.1 Sviluppi futuri . . . . . . . . . . . . . . . . . . . . . . . . 52

**Conclusioni** **53**

**Bibliografia** **55**

# Elenco delle figure


1.1 Evoluzione di alcuni linguaggi/framework dedicati al GPGPU 4

1.2 Confronto del picco teorico di performance tra CPU (Intel) e GPU (NVidia) . . . . . 5

1.3 Modello esecutivo di OpenCL . . . . . 8

1.4 Schema d'esecuzione dei work-item . . . . . 9

2.1 Differenza fra schedulazione statica e dinamica orientata alla rappresentazione degli algoritmi come Grafi Diretti Aciclici (DAG) . . . . . 16

2.2 Stack software di MAGMA . . . . . 18

2.3 Rappresentazione grafica dell'esecuzione del tile algorithm per la fattorizzazione LU del primo pannello di una matrice di $p$ = $q$ = 3 e del corrispondente aggiornamento. La matrice è di dimensioni $pb$x$qb$, dove $b$ è la dimensione di ogni *tile*. Il bordo più spesso indica i blocchi che vengono letti e il riempimento grigio indica i blocchi che vengono scritti. . . . . . 21

3.1 Varianti dell'ordine d'esecuzione degli algoritmi . . . . . 35

3.2 Varianti dell'ordine d'esecuzione del tile algorithm di fattorizzazione LU su 4 coppie CPU-GPU: le parti in fuxia sono fattorizzazioni e quelle in verde sono aggiornamenti. . . . . . 36

# Capitolo 1

# Scenario

La *Legge di Moore* ha marcato per quasi 40 anni il trend evolutivo dei microprocessori. Fino alla prima metà degli anni 2000, le prestazioni delle nuove CPU (Central Processing Unit) venivano incrementate in base al numero di transistor che era fisicamente possibile integrare nel circuito, a parità di dimensioni, e alla massima frequenza di clock raggiungibile dallo stesso. L'aumento esponenziale della frequenza nel corso dell'evoluzione di questi processi produttivi ha permesso in pratica di quadruplicare la velocità delle CPU circa ogni 3 anni, come la legge afferma.

## 1.1 Calcolo Parallelo

Con il raggiungimento dei limiti fisici e tecnologici evidenziati dall'osservazione empirica di Moore, in particolare del *power wall*[1] e dell'*ILP wall*[2], la massima frequenza cui è possibile spingere un microprocessore seriale (*Macchina di von Neumann*) si è arrestata inesorabilmente a meno di 4.0 GHz. Le aziende produttrici di microprocessori hanno investito le loro risorse nell'u-

[1] L'aumento della frequenza di un processore comporta un aumento esponenziale dell'energia necessaria al suo funzionamento (cubico rispetto alla frequenza), con conseguente abbassamento dell'efficienza rispetto a costi energetici e dissipazione del calore.

[2] L'aumento del grado di parallelismo era possibile solo a livello di istruzione ed è soggetto anch'esso a limiti di velocità ed energetici dei microprocessori.

nica via possibile per permettere ai loro prodotti di raggiungere capacità di calcolo più elevate, mantenere i consumi energetici bassi e di conseguenza rimanere competitivi sul mercato, ovvero nelle tecnologie *multicore*. Seguendo il modello dei supercalcolatori, tipicamente concepiti come sistemi composti da molteplici unità di calcolo che elaborano in concorrenza enormi quantità di dati e algoritmi particolarmente complessi, sono state prodotte e commercializzate, a partire dal 2005, nuove generazioni di microprocessori che sfruttano l'accoppiamento di due o più core che lavorano *in parallelo*, per permettere l'aumento del throughput operazionale e la distribuzione dei calcoli fra unità computazionali indipendenti.

L'aumento delle performance introdotto con il parallelismo è tuttavia bilanciato da un aumento sostanziale della complessità di scrittura dei programmi che lavorano in parallelo rispetto a quelli sequenziali, a fronte di tutte le problematiche e casistiche introdotte dall'esecuzione concorrente del codice. Inoltre, la parallelizzazione del codice è possibile solo per quei problemi che possono realmente trarre vantaggio da questo tipo di architettura e un programma ha sempre porzioni di codice sequenziale, che di fatto rallentano l'esecuzione di quelle parallele: la *Legge di Amdhal* (1967) evidenzia questo problema e formalizza l'incremento massimo della velocità d'esecuzione di un certo algoritmo in base alle proprie parti sequenziali e parallele in un sistema composto da molteplici processori.

Sono nate nel tempo molte librerie per il calcolo parallelo su sistemi multicore a memoria condivisa, come OpenMP, che hanno semplificato il processo produttivo di software che potessoro avvalersi di questo modello computazionale.

## 1.2 GPGPU

Le GPU (Graphics Processing Unit) sono dispositivi dedicati all'elaborazione e rendering di immagini. Sono nate per snellire il carico di lavoro delle CPU, accelerare le elaborazioni grafiche e progettate per lavorare su ogni

singolo pixel di un'immagine (o vertice di un modello) in maniera indipendente: la loro architettura è da sempre orientata al *parallelismo massivo* e la loro naturale evoluzione tende ad accrescere questo parallelismo, mediante la combinazione di un numero sempre maggiore di core computazionali (solitamente centinaia, molto più semplici ed economici di quelli delle CPU).

La *pipeline grafica* definisce gli stadi di elaborazione di un' immagine, da un modello *virtuale* (bidimensionali/tridimensionali) ad un' immagine *raster* (bidimensionale). In principio la pipeline era interamente implementata in hardware e poteva svolgere una serie di funzioni prefissate per ogni pixel (Fixed Function Pipeline). Con l'evolversi dell'architettura delle GPU, spinta inizialmente dalla crescita del settore video-ludico e della grafica digitale, si è passati ad un modello più flessibile che sfrutta dei programmi detti *shader*[3], in cui ogni pixel/vertice viene processato secondo un paradigma chiamato *stream processing.*

Di seguito elenchiamo le caratteristiche principali dell'architettura delle GPU moderne:

- **Latenza di memoria**: l'accesso alla memoria esterna del dispositivo (*off-chip*) è molto costoso e costituisce uno dei parametri di maggior criticità per le performance delle GPU.

- **Throughput aritmetico alto**: l'altissimo numero di ALU presenti sulle GPU e il loro design più semplice, consente al dispositivo di effettuare uno spropositato numero di operazioni aritmetiche al secondo.

- **Multi-threading massivo**: le GPU supportano centinaia di thread paralleli, grazie all'elevato numero di core indipendenti.

[3]Sono semplici programmi che descrivono i tratti di un vertice/pixel nella sua elaborazione grafica. Si distinguono in vertex shader (posizione, coordinate delle texture, colore, ecc.), pixel shader (colore, z-depth, alpha) e geometry shader (di recente introduzione, eseguono particolari primitive di composizione geometriche)

- **Bandwidth di memoria elevata**: grazie ai bus molto ampi (128 bit) e ai diversi livelli di memoria cache, l'accesso alla memoria e i trasferimenti *on-chip* sono molto veloci.

Da quando è possibile programmare gli *shader* delle pipeline grafiche, si è assistito ad un proliferare di framework e linguaggi con l'obiettivo di usare gli shader per sfruttare al massimo l'impressionante potenza di calcolo delle schede video in ambiti non esclusivamente legati alla grafica: nasce così il *General Purpose computing on Graphics Processing Unit*, abbreviato nella sigla GPGPU.

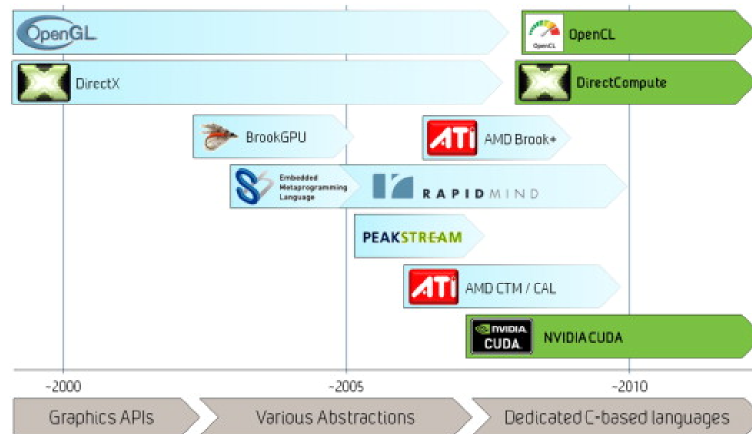



Figura 1.1: Evoluzione di alcuni linguaggi/framework dedicati al GPGPU

Il GPGPU permette di scrivere programmi in grado di avvalersi della potenza di calcolo delle GPU per problemi che possono trarre beneficio dal paradigma di *stream processing*, come algoritmi particolarmente esigenti in termini di intensità di calcolo e che possono sfruttare il parallelismo dei dati della GPU per incrementare le performance.

Uno dei primi linguaggi per questo tipo di programmazione è CUDA: fornisce un linguaggio di programmazione C-based e una serie di funzioni per scrivere applicazioni che sfruttano il calcolo parallelo dell'architettura NVidia. Attualmente CUDA è il più efficiente e ricco ambiente di sviluppo per il GPGPU: le applicazioni CUDA vengono impiegate in moltissimi ambiti

industriali e scientifici grazie al crescente numero di librerie basate su questa tecnologia, che permette di sfruttare facilmente la potenza di calcolo delle GPU NVidia per applicazioni sia grafiche che general purpose.

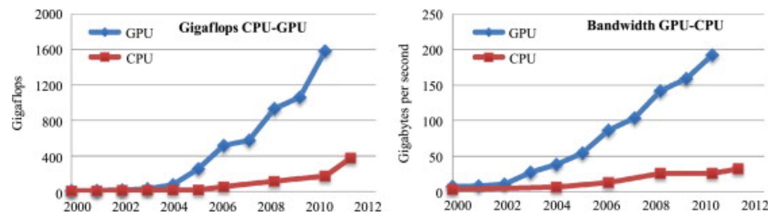



Figura 1.2: Confronto del picco teorico di performance tra CPU (Intel) e GPU (NVidia)

## 1.3 Programmazione Eterogenea

L'idea di poter sfruttare diversi dispositivi *specializzati* in precise funzioni che cooperano nel calcolo di determinati problemi, come nel caso del GPGPU, ha permesso di introdurre un nuovo metodo di programmazione conosciuto come *Programmazione Eterogenea*. Questo ha evidenziato la possibilità di scrivere programmi capaci di sfruttare le caratteristiche di diversi dispositivi, per implementare applicazioni altamente performanti e reattive.

Un *sistema eterogeneo* è un sistema composto da un *host processor*, solitamente un General Purpose Processor, che gestisce i comandi da e per diverse *unità computazionali eterogenee*. I *dispositivi eterogenei* attualmente disponibili sono *stream processor* o acceleratori hardware programmabili, in quanto dispositivi dedicati e specializzati nel calcolo intensivo o in una specifica funzione (es. CPU multicore, GPU, DSP, FPGA...). Nel caso delle GPU o CPU multicore, questi sono a loro volta composti da molteplici unità computazionali *omogenee* in grado di collaborare ad una qualche elaborazione che sfrutti data-parallelismo e task-parallelismo. Grazie ai progressi nella produzione di sistemi embedded e System-on-a-Chip (SoC), si è inoltre assistito alla nascita di piattaforme eterogenee particolarmente efficienti, composte di

processori e dispositivi di varia natura interconnessi fra loro. I sistemi AMD Fusion, Intel SandyBridge, IBM Cell, Texas Instrument OMAP e svariate piattaforme basate su ARM, come tablet o la nuova piattaforma di sviluppo NVidia CARMA, sono esempi di piattaforme eterogenee. In questo senso, ogni PC moderno è di fatto un sistema eterogeneo.

Ovviamente programmare un sistema eterogeneo non è un'operazione banale, in quanto bisogna tener conto delle caratteristiche di ogni singolo processore per garantire le massime prestazioni. Questo limita la portabilità del codice su sistemi eterogenei differenti, mostrando la necessità di un design *a blocchi* delle applicazioni, in modo da prelevare il codice specifico (precompilato o a run-time) di ogni dispositivo per adempiere ad un determinato compito. Aumenta inevitabilmente anche la difficoltà di stesura del codice: per garantire un buon rendimento, un programma dovrebbe tener conto delle differenti capacità di calcolo del dispositivo su cui viene eseguito ed essere in grado di regolare il proprio carico di lavoro in base alle sue caratteristiche. Infine, un sistema eterogeneo presenta tutte le difficoltà implementative coinvolte nella programmazione di sistemi paralleli omogenei, con l'aggiunta di tutte le problematiche relative all'uso di processori differenti fra loro: ISA[4], endianness dei dati, interfacciamento e gerarchia della memoria e tipologie di connessioni differenti per ogni dispositivo eterogeneo aumentano sostanzialmente la difficoltà di scrittura di programmi per questo tipo di sistemi.

## 1.4 OpenCL

OpenCL (Open Computing Language) è il primo standard aperto e libero da costi di licenza per la programmazione parallela di sistemi eterogenei di varia natura, come personal computer, server aziendali, sistemi integrati e SoC. OpenCL si propone come come modello efficiente di programmazio-

[4]Instruction Set Architecture

ne per incrementare velocità e interattività di varie tipologie di applicativi, utilizzando dispositivi dedicati.

L'importanza delle architetture multicore per aumentare le performance di determinati software e la difficoltà di implementazione degli stessi ha contribuito alla nascita di diverse API[5] e linguaggi ad alto livello che facilitano la stesura del codice parallelo su questi dispositivi, come OpenMP e OpenAAC. Dall'altro lato, il GPGPU e linguaggi come CUDA hanno evidenziato l'importanza della specificità del codice, per permettere un miglior utilizzo e ottenere migliori performance in base alle caratteristiche di ogni dispositivo. L'emergente necessità di applicazioni altamente efficienti, possibilmente a basso consumo e alla portata di tutti, ha permesso di far confluire questi due modelli procedurali proprio nel paradigma della Programmazione Eterogenea, e OpenCL ne è un esempio implementativo pratico.

OpenCL è attualmente gestito dal Khronos Group, un consorzio no-profit di aziende che collaborano alla definizione delle specifiche di questo (e molti altri[6]) standard e dei parametri di conformità per la creazione di driver OpenCL specifici per ogni tipo di piattaforma. Questi driver forniscono inoltre le funzioni per la compilazione dei programmi scritti lingaggio kernel: questi vengono convertiti in programmi in una qualche forma di linguaggio intermedio, solitamente vendor-specific, e poi eseguiti sulle architetture di riferimento. L'API è fornita in vari linguaggi (C, C++, Java, Python) e permette di eseguire sui device codice arbitrario o *built-in*.

Da un iniziale progetto di Apple Inc., che ne detiene i diritti per il marchio, OpenCL (attualmente alla versione 1.2) è definito come un linguaggio di programmazione basato su standard ISO C99, con opportuni costrutti computazionali che permettono di sfruttare il modello d'esecuzione parallela

[5] Application Programming Interface

[6] OpenGL, WebGL, WebCL...

SIMD[7] o SIMT[8], su cui il paradigma di *stream processing* si appoggia. In OpenCL la memoria è gestita *esplicitamente* mediante apposite funzioni e viene usato un modello di coerenza detto *relaxed consinstency model*. Insieme a questo linguaggio, viene fornita una ricca libreria di funzioni che permettono ottenere informazioni sulle piattaforme OpenCL disponibili e di gestire il trasferimento dei dati e di tutti i comandi per i dispositivi, inviati sottoforma di *kernel* scritti nel linguaggio appena descritto e compilati per il contesto esecutivo di riferimento. Il modello esecutivo di OpenCL prevede un *host processor* che gestisce una o più *heterogeneous device*.

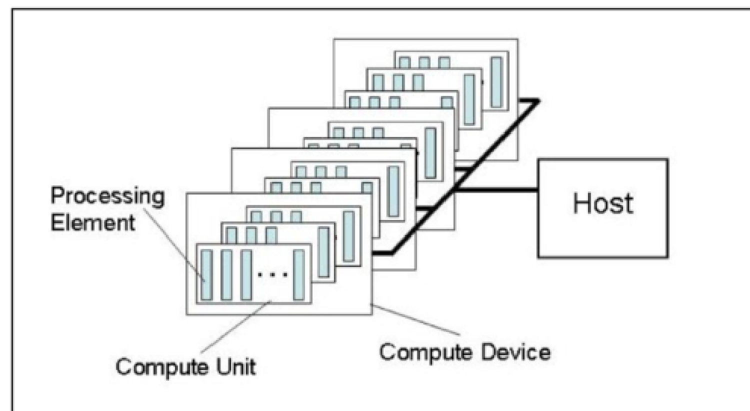



Figura 1.3: Modello esecutivo di OpenCL

Ogni comando ai device è inviato dall'host sottoforma di codice sorgente in linguaggio kernel. Il sorgente viene caricato in un *program object* (*clCreateProgramWithSource*) per l'architettura di riferimento. I program object possono essere caricati anche pre-compilati (*clCreateProgramWithBinary*) per applicazioni orientate al real-time, in modo che non vengano ricompilati durante l'esecuzione. Si procede alla compilazione (*clCreateBuild*) del

[7] Nella tassonomia computazionale dei processori, questi possono essere SISD (Single Instruction, Single Data stream), SIMD (Single Instruction, Multiple Data stream), MISD (Multiple Instruction, Single Data stream) o MIMD (Multiple Instruction, Multiple Data stream).

[8] Single Instruction, Multiple Thread

programma nell'architettura di riferimento e alla creazione del *kernel object* relativo al programma.

Un kernel object può essere eseguito in un numero di *work-group* variabile, creando una matrice di computazione *ND-imensionale* che permette di suddividere efficacemente il carico di lavoro per un problema in n-dimensioni (1,2 o 3) in ogni work-group, a sua volta composto da un numero di *work-item* che lavorano in parallelo.

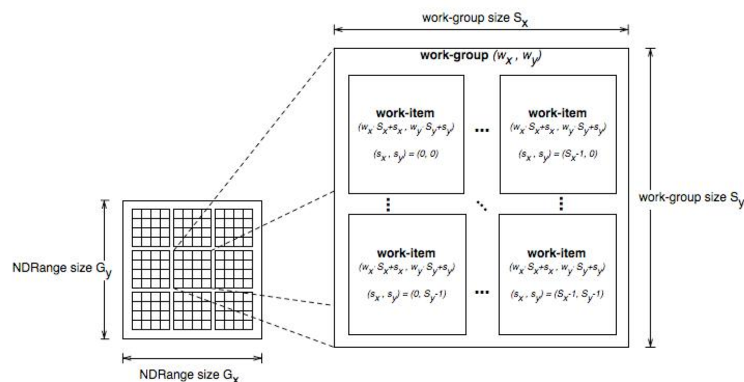



Figura 1.4: Schema d'esecuzione dei work-item

Bilanciare il carico di lavoro per ogni work-group in base alla capacità di calcolo parallelo di un dispositivo, risulta uno dei parametri critici per il raggiungimento di buone performance delle applicazioni. Un bilanciamento sbagliato del carico di lavoro, unitamente alle caratteristiche specifiche di ogni dispositivo (latenza trasferimenti, throughput, bandwidth) può portare ad una perdita sostanziale di performance o compromettere la portabilità del codice, senza considerare un qualche sistema di acquisizione dinamica di informazioni sulle capacità di calcolo dei dispositivi. Tuttavia, l'uso accurato di queste tecnologie permette di raggiungere alti livelli di performance, combinando il calcolo di diverse unità computazionali.

# Capitolo 2

# Obiettivo

L'uso di tecnologie *ibride* oltre che parallele nel calcono intensivo, come nel caso del sempre maggiore interesse verso il GPGPU in applicazioni relative all'High Parallel Computing, per via dei vantaggi in termini di prestazioni e risparmio energetico, ha permesso l'avanzamento di molti settori nel campo della ricerca scientifica e nuove opportunità di mercato.

[alcuni dati relativi ad applicazioni di CUDA e librerie, simulazioni scientifiche, applicazioni con OpenCL perchè open (non vendor specific) e orientato alla programmazione eterogenea, non solo schede video ].

Nel lavoro svolto per la mia tesi, mi sono occupato principalmente dell'uso del framework OpenCL nell'ambito applicativo dell'Algebra Lineare.

## 2.1 Algebra Lineare

Fra le possibili applicazioni del GPGPU e della Programmazione Eterogenea, il calcolo legato all'Algebra Lineare è un campo particolarmente interessante, per via della possibilità di mappare efficacemente un problema legato ad operazioni fra matrici e vettori allo schema esecutivo di *stream processing*, comune di CPU multicore e GPU. Il modello data-parallel ci permette di implementare efficacemente le funzioni primitive di queste operazioni matriciali: OpenCL (come CUDA) mette a disposizione delle operazioni vettoriali che

lavorano su agglomerati di dati e modelli di rappresentazione della memoria che permettono di sfruttare efficacemente le capacità di calcolo parallelo dei dispositivi per questo tipo di problemi. Non solo il parallelismo permette di accelerare il calcolo delle operazioni matriciali più elementari, ma la possibilità di sfruttare diverse unità computazionali che collaborano alla computazione di problemi più complessi ci porta a considerare la possibilità di progettare applicazioni particolarmente performanti, utili ad una risoluzione abbastanza efficiente per questo tipo di problemi. L'Algebra Lineare ha moltissimi ambiti applicativi, come lo studio e la simulazione di fenomeni fisici, l'analisi di sistemi economici, la rappresentazione e risoluzione di problemi legati alla teoria dei grafi, dei flussi e della crittografia.

## 2.2 MAGMA

MAGMA (Matrix Algebra on GPU and Multicore Architectures) è una libreria di funzioni di algebra lineare dedicata ai sistemi eterogenei composti da CPU multicore e GPU ed è sviluppata dal gruppo Innovative Computing Lab, presso l'Università del Tennessee. L'obiettivo di questa libreria è quello di fornire tutte le funzionalità della libreria LAPACK[1], riscritte in modo da sfruttare le capacità di calcolo delle moderne architetture ibride/eterogenee.

Il principio di funzionamento di MAGMA è molto simile a quello di LAPACK: ogni funzione si basa su funzioni più semplici, dette BLAS[2], che costituiscono lo standard *de facto* attuale per un'implementazione efficiente di funzioni di Algebra Lineare più ad alto livello. Il loro design *a blocchi* mira a sfruttare al meglio il parallelismo a livello di istruzioni (ILP), la località, il riuso dei dati e l'architettura cache-based dei microprocessori moderni, limitando il traffico di dati sui bus di memoria e garantendo buone prestazioni.

[1]Linear Algebra PACKage. Una libreria scritta in Fortran90 che fornisce funzioni efficienti per la risuluzione di problemi legati all'Algebra Lineare.

[2]Basic Linear Algebra Sub-program. Basata su 3 livelli di complessità: Liv. 1 - operazioni vettoriali; Liv. 2 - operazioni tra matrici e vettori; Liv. 3 - operazioni tra matrici.

Inoltre, per incrementare ulteriormente le performance, si è soliti utilizzare versioni di BLAS ottimizzate, in genere dalle stesse aziende produttrici, per i dispositivi su cui si vuole effettuare questo tipo di computazioni. Anche le funzioni implementate in MAGMA sono basate su versioni di BLAS ottimizzate per l'architettura di riferimento, in questo caso progettate per accelerare i calcoli usando le GPU e le CPU multicore.

Attualmente, esistono tre diverse versioni di MAGMA:

- **MAGMA 1.3**: scritta in CUDA, intesa per l'accelerazione dei calcoli esclusivamente su GPU NVidia (multiple).
- **clMAGMA 1.0**: versione OpenCL di MAGMA, attualmente intesa per l'esecuzione su GPU ATI (singola).
- **MAGMA MIC 0.3**: versione dedicata all'Intel Xeon Phi Coprocessor.

## 2.3 Algoritmi a blocchi e limiti di scalabilità

In una tipica computazione di LAPACK, la matrice e i dati su cui lavorare vengono divisi in blocchi, detti *pannelli*. L'idea della suddivisione in blocchi ruota attorno ad una importante proprietà delle funzioni BLAS Liv. 3 chiamata *surface-to-volume*: vengono eseguite $O(n^3)$ operazioni floating-point spostando una quantità di $O(n^2)$ dati. Questa proprietà permette di ridurre il trasferimento di dati e di garantirne il riuso nelle cache di alto livello, che sono anche le più veloci. Gli algoritmi basati sulla divisione in pannelli sono implementati in modo che solo una minima parte della computazione si appoggi su BLAS Liv. 2[3] e la maggior parte delle computazioni siano BLAS Liv. 3.

I due passi fondamentali di questi algoritmi possono essere descritti in questo modo:

[3]BLAS Liv. 2 non garantiscono il riuso dei dati, inquanto eseguono $O(n^2)$ operazioni floating-point su $O(n^2)$ dati. La loro implementazione è efficiente solo su CPU vettoriali.

- **fattorizzazione pannello**: viene prelevato il pannello e si applica qualche tipo di *trasformazione* relativa alla funzione che si sta eseguendo. Tipicamente, queste trasformazioni sono assimilabili a funzioni BLAS Liv. 2 e possono essere cumulabili, a seconda della funzione che si sta calcolando.

- **aggiornamento sotto-matrice**: in questo passaggio, tutte le trasformazioni effettuate nella fattorizzazione del pannello vengono applicate contemporaneamente al resto della matrice. L'aggiornamento è assimilabile a funzioni BLAS Liv. 3.

Una volta aggiornata la sotto-matrice, si ripete la fattorizzazione sul pannello successivo e si aggiorna nuovamente la sotto-matrice restante, ripetendo questo ciclo fino all'ultimo pannello disponibile.

Dato che la dimensione del pannello è molto piccola se confrontata alla relativa sotto-matrice da aggiornare, questi algoritmi a blocchi sono composti principalmente di funzioni BLAS Liv. 3, che sono le più performanti sui sistemi cache-based e traggono maggiori benefici da questo design. La computazione di funzioni BLAS Liv. 2 è invece limitata dalla velocità con cui il bus di memoria può fornire i dati ai core computazionali: nei microprocessori moderni la velocità di calcolo è nettamente superiore alla velocità dei trasferimenti e ciò significa che un solo core sarebbe sufficiente a saturare il bus. Una versione parallela di questi algoritmi a blocchi non apporterebbe nessun miglioramento sostanziale di performance, a causa dei bottleneck provocati dall'intasamento dei bus delle operazioni sequenziali. Di conseguenza un'implementazione parallela di questi algoritmi solo a livello delle funzioni BLAS (ad es. implementazioni multithreaded, accelerazione BLAS sulla GPU) indurrebbe ad un approccio al problema di tipo *fork-join*: una fattorizzazione a blocchi, a causa delle dipendenze di dati, risulterebbe in una sequenza di operazioni sequenziali (ad es. fattorizzazione dei pannelli) intervallate da operazioni parallele (ad es. l'aggiornamento della sotto-matrice). La scalabilità per problemi più grandi viene limitata dal costo delle computazioni sequenziali, che crescerebbe insieme al grado di parallelismo, e l'asincronia

dei task non sarebbe garantita, perchè i task paralleli sono forzati a *sincronizzarsi* e ad aspettare in stato IDLE la computazione di lunghi task sequenziali.

## 2.4 Approccio per sistemi multicore paralleli

L'approccio utilizzato in MAGMA per aggirare questi limiti di scalabilità per sistemi multicore consiste nello spostare il Parallelismo dal Livello di Istruzione (ILP) al Livello di Thread (TLP), seguendo l'attuale tendenza per l'incremento di performance che i sistemi eterogenei offrono.

L'approccio ottimale prevede la conversione degli algoritmi a blocchi di LAPACK in algoritmi che si adattino meglio alle caratteristiche dei sistemi multicore, chiamati *tile algorithm*. Questi permettono di suddividere la fattorizzazione dei pannelli e l'aggiornamento della sotto-matrice in task paralleli di granularità più fine. In questo modo, è possibile iniziare l'aggiornamento della sotto-matrice mentre il pannello viene ancora fattorizzato. L'idea di questi algoritmi consiste nella divisione dei pannelli in sottomatrici quadrate, chiamati *tile*. I *tile algorithm* presentano caratteristiche che è possibile sfruttare efficacemente in un sistema multicore:

- granularità fine per garantire un alto livello di parallelismo e per sfruttare la piccola dimensione delle cache
- asincronia per evitare punti di sincronizzazione
- Block Data Layout (BDL), per rappresentazioni di dati che possano garantire un accesso alla memoria efficiente
- Dynamic Data Driven Scheduler, per assicurare che ogni task in coda può essere processato appena le sue dipendenze dati sono tutte soddisfatte

Inizialmente questo design è stato utilizzato per realizzare algoritmi per il calcolo *out-of-memory* (detto anche *out-of-core*) di problemi algebrici che

prevedono strutture dati troppo grandi per la memoria disponibile [4]. La Figura 2.3 rappresenta graficamente l'esecuzione del tile algorithm per la fattorizzazione LU, mentre una descrizione dettagliata del funzionamento dei tile algorithm delle fattorizzazioni LU, QR e Cholesky è consultabile su [3].

L'algoritmo così implementato può essere rappresentato come un Grafo Aciclico Diretto (DAG), dove i nodi rappresentano task (sia fattorizzazioni che aggiornamenti) e gli archi rappresentano la dipendenza dei dati delle computazioni. L'esecuzione *sincrona* prevede la parallelizzazione del codice esclusivamente al livello di aggiornamento della matrice, utilizzando il meccanismo *fork-join* degli algoritmi a blocchi i cui punti di sincronizzazione, complici di latenze nei trasferimenti e computazioni sequenziali intense, inducono i dispositivi a lunghi periodi di inutilizzo (IDLE). L'esecuzione *asincrona* invece, basata sulla rappresentazione dell'algoritmo come un DAG, permette di rilassare la sincronizzazione delle chiamate BLAS durante l'iterazione dell'algoritmo, rispettando la dipendenza dei dati e allo stesso tempo sfruttando tutte le risorse hardware disponibili per la computazione simultanea *out-of-order* dei task. La possibilità di schedulare dinamicamente a run-time questi task, bilanciando le computazioni indipendenti sui device disponibili, apporterebbe ulteriori miglioramenti a questo tipo di computazione.

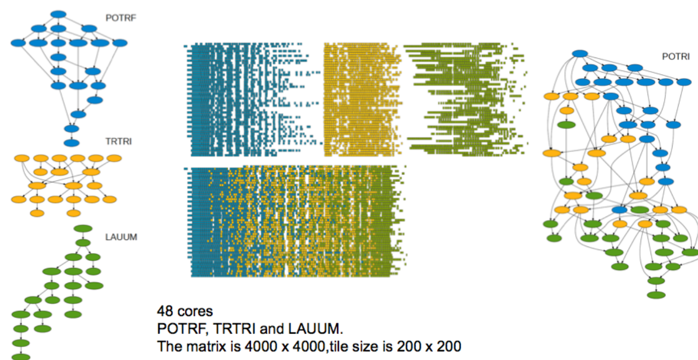



Figura 2.1: Differenza fra schedulazione statica e dinamica orientata alla rappresentazione degli algoritmi come Grafi Diretti Aciclici (DAG)

Gli algoritmi in MAGMA sono basati su un meccanismo *ibrido* fra l'approccio *fork-join* e quello a scheduling dinamico di DAG: i task sequenziali sono eseguiti sulle CPU e quelli paralleli su GPU. Lo scheduling asincrono permette di eseguire computazioni simultanee, distribuendo il carico di lavoro tra CPU multicore e GPU disponibili, e permette di trarre benefici dai sistemi eterogenei per questo tipo di algoritmi.

Anche se questo approccio può aumentare la velocità d'esecuzione su sistemi eterogenei, è esposto anch'esso a problemi di scalabilità. Questo perchè una granularità relativamente alta dei task potrebbe risultare poco flessibile per tipi diversi di device e l'esecuzione parallela dell'algoritmo può essere affetta da un degradamento delle prestazioni, dovuto al bilanciamento erroneo del workload.

## 2.5 Differenze tra MAGMA e clMAGMA

MAGMA, nella versione attuale in CUDA, mette a disposizione più di 80 diversi algoritmi (basati su un totale di oltre 320 routine standard e non) altamente performanti per la computazione ibrida di un vasta gamma di funzioni di Algebra Lineare, in diversi gradi di precisione. clMAGMA consiste essenzialmente in un porting delle routine da CUDA a OpenCL, mantenendo praticamente inalterata l'infrastruttura della libreria e l'implementazione delle funzioni.

Mentre MAGMA gode di una maggiore maturità e varietà di implementazione degli algoritmi, clMAGMA risulta essere appena più indietro nello sviluppo e differisce dalla sorella maggiore, a parte per alcune routine non ancora implementate in OpenCL, nella totale mancanza di algoritmi *out-of-memory* e multi-device. Questo è principalmente dovuto alla differente natura e modalità di interfacciamento all'hardware di CUDA e OpenCL. CUDA non richiede nessun tipo di interfacciamento specifico con l'hardware parallelo, in quanto vendor-specific (limitato all'utilizzo su GPU NVidia) e il compilatore CUDA (nvcc) produce programmi in codice PTX universalmen-

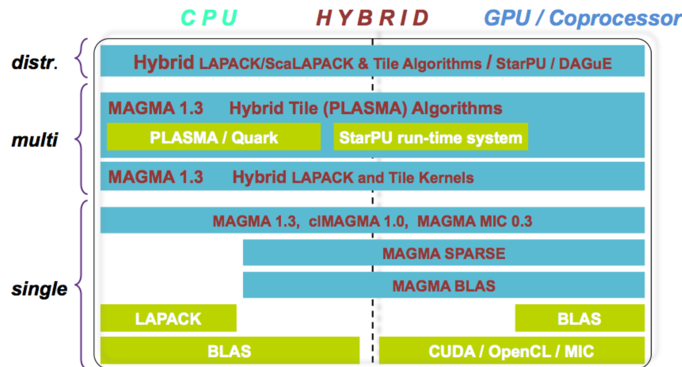



Figura 2.2: Stack software di MAGMA

te riconosciuto dai dispositivi NVidia: di conseguenza è molto facile durante la stesura del codice indirizzare le computazioni a GPU diverse, indicando semplicemente l'indice progressivo che identifica univocamente una GPU come argomento della funzione *CudaSetDevice()* e la funzione da eseguire. OpenCL, essendo cross-platform e non disponendo di un compilatore unico per tutti i dispositivi, richiede l'allocazione di specifiche risorse (piattaforma, contesti, code di comandi...) e la compilazione di ogni routine per il dispositivo di riferimento (sarebbe possibile compilare direttamente a run-time, ma si perderebbe in termini di performance): per via di questa maggiore verbosità di indirizzamento, che tuttavia fornisce un maggior controllo sull'ambiente, in clMAGMA tutte le routine sono attualmente intese per un unica GPU. Un altra differenza sostanziale tra le due librerie riguarda le performance relative alla scalabilità delle routine e all'ottimizzazione del codice: mentre in CUDA il compilatore si occupa di ottimizzare ulteriormente i programmi, con opportune tecniche volte a sfruttare al massimo l'architettura NVidia, OpenCL necessita di ottimizzazioni in base al dispositivo su cui si vuole eseguire i kernel, il che comporta un notevole aumento della difficoltà di scrivere di procedure performanti e limita la portabilità del codice.

## 2.6 Modifiche proposte per clMAGMA

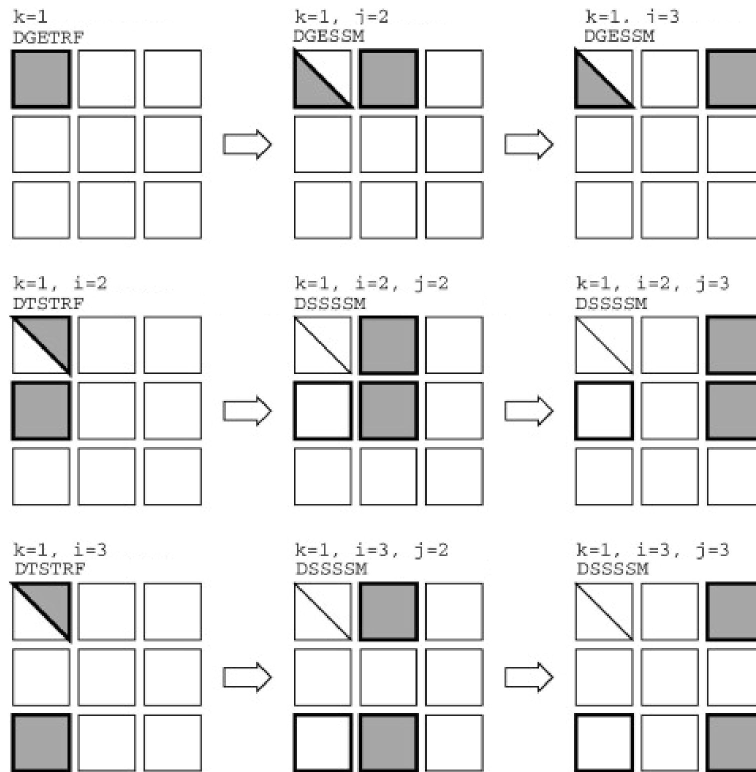
**Inizializzazione dinamica dell'ambiente di lavoro** Come già spiegato nella sezione precedente, clMAGMA 1.0 è intesa per l'utilizzo di un unico dispositivo eterogeneo, ovvero una singola GPU (ATI). OpenCL mette a disposizione diverse funzioni per l'interrogazione delle piattaforme e dispositivi eterogenei disponibili e per ottenerne informazioni rilevanti sulle loro capacità di calcolo: una procedura dinamica di inizializzazione delle risorse, sfruttando queste funzioni, permetterebbe di allocare a run-time tutte le risorse necessarie all'esecuzione dei comandi sui dispositivi eterogenei disponibili.

**Algoritmi multi-device** clMAGMA non dispone di alcun algoritmo multi-device, nè con scheduling statico, nè dinamico. Nella versione di MAGMA 1.3, gli algoritmi a scheduing dinamico basati sulle librerie QUARK e Star-PU sono ancora in fase di sviluppo (come citano su un post del forum di MAGMA), mentre quelli multi-GPU e out-of-memory sono disponibili. L'implementazione praticamente identica delle due librerie, la somiglianza delle chiamate API e le analogie del modello computazionale di CUDA e OpenCL sulle GPU, permettono di effettuare agilmente un porting di queste procedure da una libreria all'altra, utilizzando opportunamente le funzioni di basso livello delle rispettive implementazioni.

**Kernel OpenCL cross-platform** La natura cross-platform di OpenCL permette di utilizzare i kernel con cui sono implementate le routine in cl-MAGMA su diversi tipi di dispositivi eterogenei. Come già spiegato, i driver OpenCL dispongono di due tipi di funzioni di compilazione dei kernel, ovvero a run-time da sorgente oppure da program object pre-compilati per il dispositivo su cui si vuole eseguire la computazione. La differenza tra i due approcci incide sulle performance d'esecuzione di clMAGMA: mentre un approccio a run-time risulta essere di facile implementazione e permette di compilare un sorgente direttamente per l'architettura di riferimento, non manterrebbe la natura real-time della libreria e subirebbe un peggioramen-

to di performance dovuto alla compilazione dei kernel. clMAGMA dispone già di un meccanismo di generazione dei kernel pre-compilati, generati in fase di compilazione della libreria, per un dispositivo definito staticamente nel makefile. Un rilevamento dinamico dei dispositivi, proposto nel punto precedente, permetterebbe la generazione dei kernel per tutti i dispositivi utilizzabili, aumentando notevolmente la flessibilità d'utilizzo della libreria su dispositivi diversi dalle GPU, come le CPU multicore.

**Auto-bilanciamento del carico di lavoro** Tra i parametri critici delle performance dei kernel, quello del bilanciamento adeguato del workload in base alla capacità di calcolo dei dispositivi è sicuramente tra i più importanti. OpenCL offre un accesso diretto a queste informazioni, rilevando dati particolarmente interessanti per questo tipo di bilanciamento, come la quantità di memoria on-chip, la frequenza, il massimo numero di work-item eseguibili e molte altre ancora. Una tecnica automatica di calcolo del bilanciamento adeguato è facilmente implementabile utilizzando l'API OpenCL per interrogare i dispositivi stessi.




**Algorithm 3.** The tiled algorithm for LU factorization.

1. **for** $k = 1, \ldots, min(p, q)$ **do**
2. DGETRF($A_{kk}$, $L_{kk}$, $U_{kk}$, $P_{kk}$)
3. **for** $j = k + 1, \ldots, q$ **do**
4. DGESSM($A_{kj}$, $L_{kk}$, $P_{kk}$, $U_{kj}$)
5. **endfor**
6. **for** $i = k + 1, \ldots, p$ **do**
7. DTSTRF($U_{kk}$, $A_{ik}$, $P_{ik}$)
8. **for**$j = k + 1, \ldots, q$ **do**
9. DSSSSM($U_{kj}$, $A_{ij}$, $L_{ik}$, $P_{ik}$)
10. **endfor**
11. **endfor**
12. **endfor**

Figura 2.3: Rappresentazione grafica dell'esecuzione del tile algorithm per la fattorizzazione LU del primo pannello di una matrice di $p = q = 3$ e del corrispondente aggiornamento. La matrice è di dimensioni $pb$x$qb$, dove $b$ è la dimensione di ogni *tile*. Il bordo più spesso indica i blocchi che vengono letti e il riempimento grigio indica i blocchi che vengono scritti.

# Capitolo 3

# Implementazione

Tra le modifiche proposte nel capitolo precedente, andiamo ad analizzare in dettaglio l'inizializzazione dinamica dell'ambiente e l'implementazione degli algoritmi multi-device a scheduling statico. L'ambiente di test su cui si è lavorato è così formato:

- 1x CPU Intel i3 (quad-core) @ 3.07GHz - 4 GB DDR
- 1x GPU NVidia Quadro 600 (96 CUDA cores, Fermi Arch.) @ 1280MHz - 1 GB GDDR

## 3.1 Inizializzazione dinamica dell'ambiente

L'implementazione proposta per l'inizalizzazione dinamica dell'ambiente di elaborazione prevede modifiche nell'inizializzazione dell'istanza della classe CL_MAGMA_RT.

La classe nella versione originale di clMAGMA 1.0 è definita nel file interface_opencl/CL_MAGMA_RT.h in questo modo:

```
class CL_MAGMA_RT
{
private:
unsigned int MAX_GPU_COUNT;
```

```
cl_platform_id cpPlatform;
cl_uint ciDeviceCount;

cl_kernel ckKernel;             // OpenCL kernel
cl_event ceEvent;               // OpenCL event
size_t szParmDataBytes;         // Byte size of context information
size_t szKernelLength;          // Byte size of kernel code
cl_int ciErrNum;                // Error code var

bool HasBeenInitialized;
std::map<std::string, std::string> KernelMap;

int GatherFilesToCompile(const char* FileNameList, std::vector<std::string>&);
std::string fileToString(const char* FileName);
cl_device_id* cdDevices;        // OpenCL device list
cl_context cxGPUContext;        // OpenCL context
cl_command_queue *commandQueue;

CL_MAGMA_RT();   // Private constructor
~CL_MAGMA_RT();

public:

// singleton class to guarentee only 1 instance of runtime
static CL_MAGMA_RT * Instance()
{
static CL_MAGMA_RT rrt;
return &rrt;
}
cl_device_id * GetDevicePtr();
cl_context GetContext();
cl_command_queue GetCommandQueue(int queueid);
bool Init ();
bool Init(cl_platform_id gPlatform, cl_context gContext);
bool Quit ();

bool CompileFile(const char*FileName);
bool CompileSourceFiles(const char* FileNameList);
```

```
const char* GetErrorCode(cl_int err);
bool BuildFromBinaries(const char*FileName);
bool BuildKernelMap(const char* FileNameList);
bool CreateKernel(const char* KernelName);

std::map<std::string, std::string> Kernel2FileNamePool;
// kernel name -> file name
std::map<std::string, cl_program> ProgramPool; // file name -> program
std::map<std::string, cl_kernel> KernelPool; // kernel name -> kernel
};
```

L'inizializzazione a run-time di questa classe permette di allocare le risorse necessarie all'esecuzione dei kernel su un unica piattaforma *cl_platform_id* cpPlatform e un unico contesto *cl_context* cxGPUContext, relativo appunto alle sole GPU.

In una qualsiasi computazione che sfrutti questa libreria, viene effettuata la chiamata a *magma_init* presente su interface_opencl/interface.cpp, che si occupa di istanziare la classe a runtime e di inizializzare le risorse necessarie alla computazione:

```
// ========================================
// initialization
magma_err_t
magma_init()
{
    cl_int err;
    err = clGetPlatformIDs( 1, &gPlatform, NULL );
    assert( err == 0 );

    cl_device_id devices[ MagmaMaxGPUs ];
    cl_uint num;
    err = clGetDeviceIDs( gPlatform, CL_DEVICE_TYPE_GPU, MagmaMaxGPUs,
     devices, &num );
    assert( err == 0 );

    cl_context_properties properties[3] =
        { CL_CONTEXT_PLATFORM, (cl_context_properties) gPlatform, 0 };
    gContext = clCreateContext( properties, num, devices, NULL, NULL, &err );
```

```
    assert( err == 0 );

    err = clAmdBlasSetup();
    assert( err == 0 );

    // Initialize kernels related to LU
    rt = CL_MAGMA_RT::Instance();
    rt->Init(gPlatform, gContext);

    return err;
}
```

In questa funzione si ricerca 1 singola piattaforma (la prima disponibile) con la chiamata *clGetPlatformIDs*, si alloca l'array dei device *cl_device_id* e si inizializzano le sole GPU (specificato dal flag CL_DEVICE_TYPE_GPU) disponibili nella piattaforma trovata. Viene creato il contesto relativo ai dispositivi trovati e si procede all'inizializzazione delle BLAS accelerate (*clAmdBlasSetup()*). Infine, viene istanziata la classe a run-time di clMAGMA e chiamata la funzione *Init*, che si comporta in questo modo:

```
bool CL_MAGMA_RT::Init(cl_platform_id gPlatform, cl_context gContext)
{
  if (HasBeenInitialized)
    {
      printf ("Error: CL_MAGMA_RT has been initialized\n");
      return false;
    }

  printf ("Initializing clMAGMA runtime ...\n");

  cl_int ciErrNum = CL_SUCCESS;

  // set the platform
  cpPlatform    = gPlatform;

  ciErrNum  = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU, 0,
  NULL, &ciDeviceCount);
```

```
  cdDevices = (cl_device_id *)malloc(ciDeviceCount * sizeof(cl_device_id));
  ciErrNum |= clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU,
  ciDeviceCount, cdDevices, NULL);

  // set the context
  cxGPUContext = gContext;

  // create command-queues
  commandQueue = new cl_command_queue[QUEUE_COUNT];
  for(unsigned int i = 0; i < QUEUE_COUNT; i++)
    {
      // create command queue
      commandQueue[i] = clCreateCommandQueue(cxGPUContext, cdDevices[0],
     CL_QUEUE_PROFILING_ENABLE, &ciErrNum);
      if (ciErrNum != CL_SUCCESS)
{
  printf (" Error %i in clCreateCommandQueue call !!!\n\n", ciErrNum);
  return false;
}
    }
  // da qui in poi, viene riempita la mappa dei riferimenti ai kernel
  ...
  HasBeenInitialized = true;
  return true;
}
```

Si voglia notare inoltre nella classe CL_MAGMA_RT la presenza di un'ulteriore funzione *Init*, senza parametri, che viene richiamata nel sotto-programma che si occupa della pre-compilazione dei kernel durante la compilazione della libreria.

### 3.1.1 Implementazione

Le modifiche da apportare a questo tipo di chiamata devono permettere non solo di inizializzare diverse piattaforme OpenCL, ma considerare tutti i tipi di device disponibili (GPU, CPU e acceleratori) su ognuna di queste piattaforme, allocando tutte le risorse necessarie al loro utilizzo.

A tale scopo è stata dapprima creata una definizione per una struttura di dati chiamata *cl_platform*:

```
//--- MAGMA DEVICE CLASS ---

typedef struct cl_platform {
cl_platform_id  platform;
cl_uint  n_gpu;
cl_uint  n_cpu;
cl_uint  n_acc;
cl_device_id* gpu_devices; //GPU devices
cl_context   gpu_context;
cl_command_queue* gpu_queue;
cl_device_id* cpu_devices; //CPU
cl_context   cpu_context;
cl_command_queue* cpu_queue;
cl_device_id* acc_devices; //ACCELERATOR
cl_context   acc_context;
cl_command_queue* acc_queue;

cl_platform()
{
  n_gpu = n_cpu = n_acc = 0;
  platform = NULL;
  gpu_devices = NULL;
  gpu_context = NULL;
  gpu_queue = NULL;
  cpu_devices = NULL;
  cpu_context = NULL;
  cpu_queue = NULL;
  acc_devices = NULL;
  acc_context = NULL;
  acc_queue = NULL;
}
}cl_platform;
```

che definisce una struttura costituita da una serie di contatori, che indicano la quantità di risorse rilevate, puntatori a *cl_device_id* dei dispositivi e ai relativi contesti.

Alla classe CL_MAGMA_RT è stato dunque aggiunto un campo *std::vector* <*cl_platform* > cpPlatforms, un vettore C++ che andrà contenere tutte le piattaforme definite tramite la struttura *cl_platform*.

Successivamente è stata scritta una funzione *Init_all* e inserita nella classe CL_MAGMA_RT così definita:

```
bool CL_MAGMA_RT::Init_all()
{
  if (HasBeenInitialized)
    {
      printf ("Error: CL_MAGMA_RT has been initialized\n");
      return false;
    }

  printf ("Initializing clMAGMA runtime ...\n\n");

  cl_int ciErrNum = CL_SUCCESS;
  char chBuffer[1024];
  cl_platform_id* clPlatformIDs;
  cl_uint n_platform;
  cl_int index = 0;

  // set the platform


  ciErrNum = clGetPlatformIDs (0, NULL, &n_platform);

  if (ciErrNum != CL_SUCCESS){
    printf(" Error %i in clGetPlatformIDs Call !!!\n\n", ciErrNum);
  }else if(n_platform == 0){
    printf("No OpenCL platform found!\n\n");
    return false;
  //}else if ((clPlatformIDs =
  (cl_platform_id*) malloc(n_platform * sizeof(cl_platform_id))) != NULL) {
  }else if((clPlatformIDs = new cl_platform_id[n_platform]) != NULL){
    printf("-> Found %d OpenCL platform\n\n", n_platform);
    // get platform info for each platform and trap the NVIDIA platform if found
    ciErrNum = clGetPlatformIDs (n_platform, clPlatformIDs, NULL);
```

```
    for(cl_uint i = 0; i < n_platform; i++){

      ciErrNum = clGetPlatformInfo (clPlatformIDs[i], CL_PLATFORM_NAME,
       1024, &chBuffer, NULL);
      if(ciErrNum == MAGMA_SUCCESS){
printf("[%d] %s\t",i, chBuffer);
      }else{
printf("\t Error %i in clGetPlatformInfo Call !!!\n\n", ciErrNum);
return false;
      }
      ciErrNum = clGetPlatformInfo (clPlatformIDs[i], CL_PLATFORM_VERSION,
       1024, &chBuffer, NULL);
      if (ciErrNum != MAGMA_SUCCESS){
printf("\t Error %i in clGetPlatformInfo Call !!!\n\n", ciErrNum);
return false;
      } else {
  printf(" CL_PLATFORM_VERSION: \t%s\n\n", chBuffer);
  // Get and log OpenCL device info
  // save platform
  index = initPlatform(clPlatformIDs[i]);

  if(index < 0){
    printf("Error during initPlatform(): %d\n", index);
    return false;
  }
      }
    }// end platform for()
    delete [] clPlatformIDs;


  }else {
    printf("Failed to allocate memory for cl_platform ID's!\n\n");
    return false;
  }

  // set the default platform and context
  cl_platform platform;
  if(cpPlatforms.size() > 0){
```

```
    platform = cpPlatforms.at(0);
    cpPlatform    = platform.platform;
    cxGPUContext  = platform.gpu_context;
    commandQueue  = platform.gpu_queue;
    cdDevices = platform.gpu_devices;
    ciDeviceCount = platform.n_gpu;
  }else{
    printf("No OpenCL platforms found!");
    return false;
  }
  // da qui in poi, viene riempita la mappa dei riferimenti ai kernel
  ...
  HasBeenInitialized = true;
  return true;
}
```

Per ogni piattaforma rilevata dall'API di OpenCL, chiamiamo la funzione *initPlatform*:

```
//init platform, returns the index reference to cl_platform element that will
be stored in cpPlatforms class vector
cl_int CL_MAGMA_RT::initPlatform(const cl_platform_id src_platform){

  cl_platform platform;
  uint index = 0;
  std::vector<cl_platform>::iterator it;
  cl_int ciErrNum = 0;

  if(!cpPlatforms.empty()){
    cl_platform current;
    for(cl_int i = 0; i < cpPlatforms.size(); i++){
current = cpPlatforms.at(i);
if(current.platform == src_platform){
    printf("Platform %d already stored. Nothing to do.\n", i);
    return i;

    }
    }
  }
```

```
  platform.platform = src_platform;
  //init GPUS
  printf("OpenCL GPU Device Info:\n");
  printf("------------------------\n\n");
  if(!(initDevices(src_platform, &platform.gpu_devices, &platform.gpu_context,
   &platform.n_gpu, &platform.gpu_queue, CL_DEVICE_TYPE_GPU,
    MagmaMaxGPUs, &ciErrNum, "GPU"))){
    printf("Error! initDevices failed: %d\n", ciErrNum);
    return MAGMA_ERR_UNKNOWN;
  }
  printf("\n--------------------------------\n");
  //init CPUS
  printf("OpenCL CPU Device Info:\n");
  printf("------------------------\n\n");
  if(!(initDevices(src_platform, &platform.cpu_devices, &platform.cpu_context,
   &platform.n_cpu, &platform.cpu_queue, CL_DEVICE_TYPE_CPU,
    MagmaMaxDEVs, &ciErrNum, "CPU"))){
    printf("Error! initDevices failed: %d\n", ciErrNum);
    return MAGMA_ERR_UNKNOWN;
  }
  printf("\n--------------------------------\n\n");
  //push platform into global vector
  cpPlatforms.push_back(platform);
  // iterator to vector element:
  it = std::find_if(cpPlatforms.begin(),cpPlatforms.end(),
  FindPlatformID(src_platform));
  if(it != cpPlatforms.end()) {
    index = it - cpPlatforms.begin();
  }
  return index;
}
```

che si occupa di allocare le risorse relative al contesto e ai dispositivi trovati, dividendoli per tipo. In questo caso vengono reperite device di tipo GPU o CPU multicore. Infine si salva la struttura completa di tutte le informazioni nel vettore cpPlatforms, che le conterrà durante l'esecuzione a run-time. Chiudendo la chiamata di *Init_all*, vengono anche impostate la piattaforma

e la lista dei dispositivi di default da utilizzare nelle computazioni.

Di seguito riportiamo l'output di una tipica esecuzione di clMAGMA con l'inizializziazione proposta, che mostra nella nostra macchina test le piattaforme e i device rilevati:

```
Initializing clMAGMA runtime ...

-> Found 2 OpenCL platform

[0] NVIDIA CUDA  CL_PLATFORM_VERSION:  OpenCL 1.1 CUDA 4.2.1

OpenCL GPU Device Info:
------------------------

 1 GPU devices found supporting OpenCL:

devices 0x7fff6c5fa048, *devices 0xe48070, device_type 0x4, max_ndev 0x4
- GPU Device Quadro 600
- GPU Device Global Mem Size: 1073283072
- GPU Device Max Alloc Mem Size: 268320768

--------------------------------
OpenCL CPU Device Info:
------------------------

 No CPU devices in context found supporting OpenCL (return code -1)

--------------------------------

[1] AMD Accelerated Parallel Processing  CL_PLATFORM_VERSION:  OpenCL 1.2 AMD-APP (923.1)

OpenCL GPU Device Info:
------------------------

 No GPU devices in context found supporting OpenCL (return code -1)

--------------------------------
OpenCL CPU Device Info:
```

```
------------------------

 1 CPU devices found supporting OpenCL:

devices 0x7fff6c5fa060, *devices 0xec0fa0, device_type 0x2, max_ndev 0x4
- CPU Device Intel(R) Core(TM) i3 CPU         540  @ 3.07GHz
- CPU Device Global Mem Size: 4003463168
- CPU Device Max Alloc Mem Size: 2147483648

--------------------------------
```

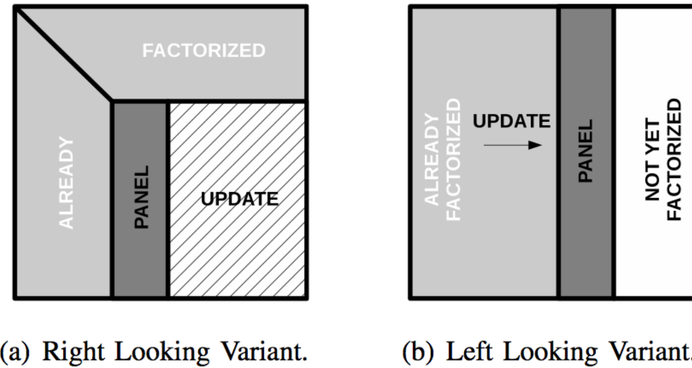
### 3.1.2 Sviluppi futuri

Ridefinendo appropriatamente le funzioni ad alto livello, con questa implementazione è possibile ad esempio indirizzare due computazioni indipendenti su due differenti piattaforme, per sfruttare la computazione simultanea, oppure distribuire la computazione di un problema molto grande su piattaforme e dispositivi eterogenei disponendo di algoritmi *out-of-memory* o multi-device.

## 3.2 Algoritmi multi-device

Le versioni implementate in clMAGMA degli algoritmi a blocchi ibridi dedicati alle fattorizzazioni LU, QR e Cholesky sono intesi per CPU e una singola GPU. Come già accennato, clMAGMA affronta questo tipo di problemi utilizzando LAPACK per la fattorizzazione (BLAS Liv. 2) dei pannelli sulla CPU e kernel OpenCL per l'aggiornamento delle sotto-matrici (BLAS Liv. 3) sulla GPU.

Un fattore critico, già riscontrato negli algoritmi a blocchi di LAPACK, riguarda l'ordine con cui il loop iterativo fattorizzazzione-aggiornamento viene eseguito: una variante *right-looking* consiste nell'aggiornare la sotto-matrice, a destra del pannello, prima che inizi la fattorizzazione del pannello suc-




(a) Right Looking Variant. (b) Left Looking Variant.

Figura 3.1: Varianti dell'ordine d'esecuzione degli algoritmi

cessivo. Questa variante genera molti task concorrenti che possono essere eseguiti potenzialmente in parallelo. La variante *left-looking*, applica aggiornamenti sequenziali generati dalla fattorizzazione del pannello precedente, prima di proseguire con la fattorizzazione del prossimo pannello. Questa variante permette un miglior riuso dei dati, ma può generare limitazioni del parallelismo.

L'esecuzione statica degli algoritmi a blocchi in LAPACK impone un'ordine lineare di scheduling dei task: possono verificarsi situazioni in cui ad esempio la GPU attende la conclusione di task sequenziali della CPU anche se i dati su cui lavorare sono già disponibili. La figura Figura 3.2 mostra i punti di stallo (spazi bianchi) del tile algorithm per la fattorizzazione LU nelle varianti RL e LL.

Grazie a queste ottimizzazioni e alla possibilità di *sovrapporre* le computazioni eterogenee, clMAGMA riesce a incrementare notevolmente le performance degli algoritmi ibridi rispetto a quelli classici di LAPACK, avvalendosi dell'uso combinato di processori di diversa natura.

Nonostante le infrastrutture di MAGMA e clMAGMA siano sostanzialmente identiche, come lo stack software delle due librerie evidenzia, la diversa natura implementativa (CUDA o OpenCL) incide sulle prestazioni dei rispettivi algoritmi. In generale, le performance di MAGMA sono migliori rispetto

(a) Right Looking (RL).

(b) Left Looking (LL).

Figura 3.2: Varianti dell'ordine d'esecuzione del tile algorithm di fattorizzazione LU su 4 coppie CPU-GPU: le parti in fuxia sono fattorizzazioni e quelle in verde sono aggiornamenti.

a quelle di clMAGMA, a parità di hardware utilizzato e implementazione degli algoritmi: questo è principalmente dovuto all'ottimizzazione del codice a cui le routine in CUDA sono sottoposte per sfruttare al meglio le caratteristiche architetturali delle GPU NVidia, mentre le performance di OpenCL sono migliorabili solo con una specifica ottimizzazione del codice per l'hardware di riferimento o con la realizzazione di driver, e relativo compilatore, più efficienti per i dispositivi che si vogliono utilizzare.

Di seguito riportiamo gli output relativi all'algoritmo ibrido di fattorizzazione LU (*sgetrf_gpu*), entrambi eseguiti sulla macchina test:

- **clMAGMA 1.0.0**:

```
[xxx@yyy clmagma-1.0.0]$ testing/testing_sgetrf_gpu

Initializing clMAGMA runtime ...

  M     N     CPU GFlop/ (sec)s   GPU GFlop/s (sec)   ||PA-LU||/(||A||*N)
=========================================================================
 1024  1024   10.67 (  0.07)      21.92 (  0.03)      1.905026e-09
 2048  2048   11.64 (  0.49)      31.30 (  0.18)      1.857872e-09
 3072  3072   11.96 (  1.62)      35.35 (  0.55)      1.762410e-09
 4032  4032   12.39 (  3.53)      36.88 (  1.18)      1.803087e-09
 4992  4992   12.60 (  6.58)      38.64 (  2.15)      1.757683e-09
```

```
 5952  5952   12.46 ( 11.28)      39.18 (  3.59)      1.754452e-09
 7104  7104   12.68 ( 18.85)      39.74 (  6.01)      1.740250e-09
 8064  8064   12.72 ( 27.48)      39.39 (  8.87)      1.862364e-09
 9000  9000   12.71 ( 38.22)      40.75 ( 11.92)      2.055784e-09
10000 10000   12.74 ( 52.32)      40.93 ( 16.29)      2.252671e-09
```

- **MAGMA 1.2.1**:

```
[xxx@yyy magma-1.2.1]$ testing/testing_sgetrf_gpu
device 0: Quadro 600, 1280.0 MHz clock, 1023.6 MB memory, capability 2.1

  M     N   CPU GFlop/s    GPU GFlop/s   ||PA-LU||/(||A||*N)
============================================================
  960   960   10.91          23.56         2.333238e-09
 1920  1920   11.87          45.27         2.012205e-09
 3072  3072   12.26          57.85         1.867271e-09
 4032  4032   12.48          75.45         2.252640e-09
 4992  4992   12.45          82.52         2.131239e-09
 5952  5952   12.72          85.69         2.059188e-09
 7104  7104   12.80          87.79         1.964651e-09
 8064  8064   12.82          89.55         2.095500e-09
 9024  9024   12.85          90.87         2.268683e-09
 9984  9984   12.86          91.77         2.422947e-09
```

Nonostante l'hardware utilizzato e le due versioni del codice siano praticamente identiche (cambiano solo le chiamate alle funzioni di basso livello), la sostanziale differenza di performance esalta immediatamente la maturità di MAGMA, e quindi di CUDA, rispetto alla versione OpenCL. Nonostante questa differenza, è importante notare il notevole guadagno prestazionale che gli algoritmi ibridi permettono di raggiungere, confrontando le performance degli algoritmi di LAPACK rispetto a quelli ibridi di MAGMA / clMAGMA.

### 3.2.1 Implementazione

MAGMA, come già accennato, dispone anche di algoritmi ibridi multi-GPU: l'idea alla base della loro versione multi-device consiste nella divisione della matrice in pannelli di $nb$ colonne, ognuno dei quali è distribuito ciclicamente sulle GPU disponibili. La fattorizzazione avviene seguendo il classico approccio fattorizzazione-aggiornamento con look-ahead dinamico (LL): per ogni pannello, la GPU che detiene il pannello corrente, lo invia alla CPU; la CPU effettua la fattorizzazione; con look-ahead si assegna alla GPU attuale il prossimo pannello da fattorizzare non ancora assegnato; le GPU aggiornano in parallelo la matrice e si ripete il tutto per il pannello successivo.

Mostriamo ora il codice della routine ibrida di fattorizzazione LU multi-GPU implementato per clMAGMA, sulla base della versione dell'algoritmo presente su MAGMA 1.2.1:

```
/*
    Purpose
    =======

    SGETRF computes an LU factorization of a general M-by-N matrix A
    using partial pivoting with row interchanges.

    The factorization has the form
       A = P * L * U
    where P is a permutation matrix, L is lower triangular with unit
    diagonal elements (lower trapezoidal if m > n), and U is upper
    triangular (upper trapezoidal if m < n).

    This is the right-looking Level 3 BLAS version of the algorithm.

    Arguments
    =========

    NUM_GPUS
            (input) INTEGER
            The number of GPUS to be used for the factorization.
```

```
    M       (input) INTEGER
            The number of rows of the matrix A.  M >= 0.

    N       (input) INTEGER
            The number of columns of the matrix A.  N >= 0.

    A       (input/output) REAL array on the GPU, dimension (LDDA,N).
            On entry, the M-by-N matrix to be factored.
            On exit, the factors L and U from the factorization
            A = P*L*U; the unit diagonal elements of L are not stored.

    LDDA     (input) INTEGER
            The leading dimension of the array A.  LDDA >= max(1,M).

    IPIV    (output) INTEGER array, dimension (min(M,N))
            The pivot indices; for 1 <= i <= min(M,N), row i of the
            matrix was interchanged with row IPIV(i).

    INFO    (output) INTEGER
            = 0:  successful exit
            < 0:  if INFO = -i, the i-th argument had an illegal value
                  or another error occured, such as memory allocation failed.
            > 0:  if INFO = i, U(i,i) is exactly zero. The factorization
                  has been completed, but the factor U is exactly
                  singular, and division by zero will occur if it is used
                  to solve a system of equations.
    =====================================================================    */

#include <math.h>
#include "common_magma.h"
#include "../testing/testings.h"

#define inAT(id,i,j) d_lAT[(id)], ((i)*nb*lddat + (j)*nb)

magma_int_t
magma_sgetrf_mgpu(magma_int_t num_gpus,
                 magma_int_t m, magma_int_t n,
                 magmaFloat_const_ptr *d_lA, magma_int_t ldda,
```

```
                 magma_int_t *ipiv, magma_queue_t *queue, magma_int_t *info)
{
    float c_one     = MAGMA_S_MAKE(  1.0, 0.0 );
    float c_neg_one = MAGMA_S_MAKE( -1.0, 0.0 );

    magma_int_t iinfo, nb, n_local[num_gpus], ldat_local[num_gpus];
    magma_int_t maxm, mindim;
    magma_int_t i, j, d, rows, cols, s, lddat, lddwork, ldpan[num_gpus];
    magma_int_t id, i_local, i_local2, nb0, nb1;
    magma_ptr d_lAT[num_gpus], d_lAP[num_gpus], d_panel[num_gpus], panel_local[num_gpus];
    float *work;
    magma_event_t event[num_gpus];

    magma_err_t err;

    //Check arguments
    *info = 0;
    if (m < 0)
        *info = -2;
    else if (n < 0)
        *info = -3;
    else if (ldda < max(1,m))
        *info = -5;

    if (*info != 0) {
        magma_xerbla( __func__, -(*info) );
        return *info;
    }

    /* Quick return if possible */
    if (m == 0 || n == 0)
        return *info;

    /* Function Body */
    mindim = min(m, n);
    nb     = magma_get_sgetrf_nb(m);

    if (nb <= 1 || nb >= n) {
```

```
//Usa solo CPU
err = magma_malloc_host( (void**) &work, m*n*sizeof(float) );
        if ( err != MAGMA_SUCCESS ) {
          *info = MAGMA_ERR_HOST_ALLOC;
          return *info;
        }
        chk( magma_sgetmatrix( m, n, d_lA[0], 0, ldda, work, 0, m, queue[0] ));
    //fattorizzo intera matrice con LAPACK
        lapackf77_sgetrf(&m, &n, work, &m, ipiv, info);
        chk( magma_ssetmatrix( m, n, work, 0, m, d_lA[0], 0, ldda, queue[0] ));
free(work);
    } else {
//Usa algoritmo ibrido CPU-GPU
        maxm = ((m + 31)/32)*32;

        if( num_gpus > ceil((float)n/nb) ) {
          printf( " * too many GPUs for the matrix size, using %d GPUs\n", (int) num_gpus );
          *info = -1;
          return *info;
        }

        /* allocazione della memoria necessaria alle GPU */
        for(i=0; i<num_gpus; i++){


  /* local-n e local-ld */
  n_local[i] = ((n/nb)/num_gpus)*nb;

          if (i < (n/nb)%num_gpus) n_local[i] += nb;
  else if (i == (n/nb)%num_gpus) n_local[i] += n%nb;

          ldat_local[i] = ((n_local[i]+31)/32)*32;

  /* workspaces */
          if (MAGMA_SUCCESS != magma_malloc( &d_lAP[i], nb*maxm*sizeof(float) )) {
    for( j=0; j<i; j++ ) {

              magma_free( d_lAP[j]   );
```

```
            magma_free( d_panel[j] );
            magma_free( d_lAT[j]   );
          }
          *info = MAGMA_ERR_DEVICE_ALLOC;
          return *info;
        }

        if (MAGMA_SUCCESS != magma_malloc( &d_panel[i], nb*maxm*sizeof(float) )) {
  for( j=0; j<=i; j++ ) {
            magma_free( d_lAP[j] );
          }
          for( j=0; j<i; j++ ) {
            magma_free( d_panel[j] );
            magma_free( d_lAT[j]   );
  }
          *info = MAGMA_ERR_DEVICE_ALLOC;
          return *info;
        }

        /* local-matrix storage */
        lddat = ldat_local[i];

        if (MAGMA_SUCCESS != magma_malloc( &d_lAT[i], lddat*maxm*sizeof(float) )) {
  for( j=0; j<=i; j++ ) {
            magma_free( d_lAP[j]   );
            magma_free( d_panel[j] );
          }
          for( j=0; j<i; j++ ) {
            magma_free( d_lAT[j] );
          }
          *info = MAGMA_ERR_DEVICE_ALLOC;
          return *info;
        }

        magma_stranspose2(d_lAT[i], 0, lddat, d_lA[i], 0,  ldda, m, n_local[i], queue[i]);
magma_sub_buffer(&panel_local[i], inAT(i,0,0));

ldpan[i] = lddat;
```

```
}

// Alloca l'area per il pannello sulla cpu workspace
lddwork = maxm;
if(MAGMA_SUCCESS != magma_malloc_host((void**) &work, lddwork*nb*sizeof(float))){
  for(i=0; i < num_gpus; i++){
    magma_free(d_lAP[i]   );
            magma_free( d_panel[i] );
            magma_free( d_lAT[i]   );
  }
  *info = MAGMA_ERR_HOST_ALLOC;
  return *info;
}


s = mindim/nb;

// per ogni pannello della matrice

for(i=0; i<s; i++){

  // Setta l'id della GPU che controlla il pannello corrente
  id = i%num_gpus;
  //Setta l'indice locale del pannello
  i_local = i/num_gpus;
  cols = maxm - i*nb;
  rows = m - i*nb;
  lddat = ldat_local[id];

  magma_stranspose(d_lAP[id], 0, cols, inAT(id,i,i_local), lddat, nb,
  cols, queue[id] );
  //copiamo (asincronamente) d_lAP[id] dalla GPU a work (CPU) per la fattorizzazione
  magma_sgetmatrix_async(rows, nb, d_lAP[id], 0, cols, work, 0, lddwork,
   queue[id], &event[id]);

  //Ci si assicura che la queue della id-GPU sia vuota
  magma_event_sync(event[id]);
```

```
// NOTE: dalla seconda iterazione (i > 0) Aggiornamento matrice
if(i > 0){
  magma_strsm(MagmaRight, MagmaUpper, MagmaNoTrans, MagmaUnit,
   n_local[id]-(i_local+1)*nb, nb, c_one, panel_local[id], 0, ldpan[id],
   inAT(id,i-1,i_local+1), lddat, queue[id]);

  magma_sgemm( MagmaNoTrans, MagmaNoTrans,
                 n_local[id]-(i_local+1)*nb, rows, nb,
                  c_neg_one, inAT(id,i-1,i_local+1),           lddat,
                 panel_local[id], nb*ldpan[id], ldpan[id],
                  c_one,     inAT(id,i,  i_local+1),           lddat, queue[id] );

}

//sincronizziamo la fattorizzazione dell'i-esimo pannello con la id-GPU
//per l'eventuale aggiornamento
magma_queue_sync(queue[id]);

// Fattorizzazione i-esimo pannello
    lapackf77_sgetrf( &rows, &nb, work, &lddwork, ipiv+i*nb, &iinfo);

if ( (*info == 0) && (iinfo > 0) ) {
//errore nella fattorizzazione
  *info = iinfo + i*nb;
          break;
        }

    // Inviamo il pannello a TUTTE le GPU
for( d=0; d<num_gpus; d++ ) { //per ogni GPU
  lddat = ldat_local[d];
  //copia il pannello fattorizzato in ogni GPU nell'area d_lAP[d]
  magma_ssetmatrix_async( rows, nb, work, 0, lddwork, d_lAP[d], 0, maxm,
   queue[d], &event[d] );
        }
```

```
      for( d=0; d<num_gpus; d++ ) { //per ogni GPU

lddat = ldat_local[d];
//applica pivoting alla copia locale della matrice
if( d == 0 )
  magma_spermute_long2(lddat, d_lAT[d], 0, lddat, ipiv, nb, i*nb, queue[d]);
else
  magma_spermute_long3(d_lAT[d], 0, lddat, ipiv, nb, i*nb, queue[d]);

// se il pannello fattorizzato appartiene a questa GPU
if(d == id){
  magma_sub_buffer(&panel_local[d], inAT(d,i,i_local));
  ldpan[d] = lddat;
  // colonna successiva
  i_local2 = i_local+1;



} else {
  // il pannello appartiene ad un altra gpu
  panel_local[d] = d_panel[d];
  ldpan[d] = nb;
  // colonna successiva
  i_local2 = i_local;
  if( d < id ) i_local2 ++;

}

// dimensione del prossimo pannello
if(s > (i+1)){ //se la prossima iterazione non Ã¨ l'ultima
  nb0 = nb; //la colonna sarÃ  sempre di dimensione nb
} else {
  nb0 = n_local[d]-nb*(s/num_gpus); //altrimenti
  if(d < s%num_gpus) nb0 -= nb;
}

if(d == (i + 1)%num_gpus){
  // ottiene la prossima colonna e fa look-ahead
  nb1 = nb0;
```

```
    }else {
      //aggiorna la matrice
      nb1 = n_local[d] - i_local2*nb;
    }

    // sincronizzazione
    magma_queue_sync(queue[d]);
    magma_stranspose2(panel_local[d], 0, ldpan[d], d_lAP[d], 0, maxm,
    cols, nb, queue[d]);
    // GPU aggiorna la matrice
    magma_strsm(MagmaRight, MagmaUpper, MagmaNoTrans, MagmaUnit,
    nb1, nb, c_one, panel_local[d], 0, ldpan[d],
    inAT(d, i, i_local2), lddat, queue[d]);

    magma_sgemm(MagmaNoTrans, MagmaNoTrans, nb1, m-(i+1)*nb, nb,
     c_neg_one, inAT(d, i,   i_local2), lddat, panel_local[d], nb*ldpan[d],
     ldpan[d], c_one, inAT(d, i+1, i_local2), lddat, queue[d]);



  } //Fine aggiornamenti GPU

} //Fine for i=1..s

// ultimo pannello

// Setta l'id dell'ultima GPU che lavora sull'ultimo pannello
          id = s%num_gpus;

          // Setta l'indice locale dell'ultimo pannello
          i_local = s/num_gpus;

          // dimensione dell'ultimo blocco-diagonale
          nb0 = min(m - s*nb, n - s*nb);
          rows = m    - s*nb;
          cols = maxm - s*nb;
          lddat = ldat_local[id];
```

```
  if( nb0 > 0 ) {
    // Invio dell'ultimo pannello alla CPU
    magma_stranspose2(d_lAP[id], 0, maxm, inAT(id,s,i_local), lddat, nb0,
     rows, queue[id]);
    magma_sgetmatrix(rows, nb0, d_lAP[id], 0, maxm, work, 0, lddwork, queue[id]);

    magma_queue_sync(queue[id]);
    // fattorizzazione sulla CPU
            lapackf77_sgetrf( &rows, &nb0, work, &lddwork, ipiv+s*nb, &iinfo);
            if ( (*info == 0) && (iinfo > 0) ) *info = iinfo + s*nb;

    // invia l'ultimo pannello fattorizzato a tutte le GPU
            for( d=0; d<num_gpus; d++ ) {
              lddat = ldat_local[d];
              i_local2 = i_local;
              if( d < id ) i_local2 ++;

              if( d == id || n_local[d] > i_local2*nb ){
magma_ssetmatrix_async(rows, nb0, work, 0, lddwork, d_lAP[d], 0, maxm,
 queue[d], &event[d]);
      }
    }
}
// clean up
for( d=0; d<num_gpus; d++ ) {

    lddat = ldat_local[d];

  if( nb0 > 0 ) {
    if( d == 0 )
      magma_spermute_long2(lddat, d_lAT[d], 0, lddat, ipiv, nb0, s*nb, queue[d]);
    else
      magma_spermute_long3(d_lAT[d], 0, lddat, ipiv, nb0, s*nb, queue[d]);

    i_local2 = i_local;
            if( d < id ) i_local2++;

            if( d == id ) {
```

```
      // il pannello appartiene a questa GPU
      err = magma_sub_buffer(&panel_local[d], inAT(d,s,i_local) );
      // setta la prossima nb-colonna
      nb1 = n_local[d] - i_local*nb-nb0;

      magma_queue_sync(queue[d]);
      magma_stranspose2(panel_local[d], 0, lddat, d_lAP[d], 0, maxm, rows, nb0,
       queue[d]);

  if(nb1 > 0)
    magma_strsm(MagmaRight, MagmaUpper, MagmaNoTrans, MagmaUnit,
nb1, nb0, c_one,
panel_local[d], 0, lddat,
inAT(d,s,i_local2)+nb0, lddat, queue[d]);

      } else if(n_local[d] > i_local2*nb){
// il pannello appartiene ad un altra GPU
                panel_local[d] = d_panel[d];

                // prossimo pannello
                nb1 = n_local[d] - i_local2*nb;

magma_queue_sync(queue[d]);
magma_stranspose2( panel_local[d], 0, nb0, d_lAP[d], 0, maxm, rows, nb0, queue[d]);
magma_strsm(MagmaRight, MagmaUpper, MagmaNoTrans, MagmaUnit,
nb1, nb0, c_one, panel_local[d], 0, nb0,
                 inAT(d,s,i_local2) , lddat, queue[d]);
      }
    }
    // salva e termina
    magma_stranspose2(d_lA[d], 0, ldda, d_lAT[d], 0, lddat, n_local[d], m, queue[d]);
    magma_queue_sync(queue[d]);
    magma_free( d_lAT[d]   );
           magma_free( d_lAP[d]   );
    magma_free( panel_local[d]);
           magma_free( d_panel[d] );
```

```
}
magma_free_host(work);

    }
    return *info;
}
#undef inAT
```

Per scrivere questo algoritmo sono state aggiunte alcune funzioni precedentemente non implementate in clMAGMA, come le funzioni di creazione di sub-buffer e funzioni di gestione degli eventi OpenCL.

Non disponendo di una macchina con multiple GPU, si è potuto testare il funzionamento di questo algoritmo eseguendolo solo sulla GPU singola della macchina test.

Riportiamo di seguito gli output relativi ai due algoritmi ibridi di fattorizzazione LU per multi GPU, ma eseguiti su GPU singola, delle rispettive librerie:

- **clMAGMA 1.0.0**:

```
[xxx@yyy clmagma-1.0.0]$ testing/testing_sgetrf_gpu

Initializing clMAGMA runtime ...

  M     N   CPU GFlop/s    GPU GFlop/s   ||PA-LU||/(||A||*N)
============================================================
  960   960   10.68          13.48         1.116691e-01
 1920  1920   11.61          25.54         1.453352e-01
 3072  3072   11.98          30.46         1.720732e-01
 4032  4032   12.19          32.08         1.844489e-01
 4992  4992   12.33          34.18         1.989220e-01
 5952  5952   12.58          35.27         2.178062e-01
 7104  7104   12.67          36.26         2.342581e-01
 8064  8064   12.66          36.37         2.608660e-01
 9024  9024   12.77          37.29         3.031985e-01
 9984  9984   12.76          37.64         3.441655e-01
```

- **MAGMA 1.2.1**:

```
[xxx@yyy magma-1.2.1]$ testing/testing_sgetrf_mgpu
device 0: Quadro 600, 1280.0 MHz clock, 1023.6 MB memory, capability 2.1

  M     N   CPU GFlop/s    GPU GFlop/s   ||PA-LU||/(||A||*N)
============================================================
  960   960   10.59          24.09         2.333238e-09
 1920  1920   11.58          45.33         2.012205e-09
 3072  3072   12.09          58.20         1.867271e-09
 4032  4032   12.20          77.16         2.252640e-09
 4992  4992   12.47          82.20         2.131239e-09
 5952  5952   12.67          85.48         2.059188e-09
 7104  7104   12.71          87.69         1.964651e-09
 8064  8064   12.75          89.38         2.095500e-09
 9024  9024   12.69          90.66         2.268683e-09
 9984  9984   12.80          91.68         2.422947e-09
```

Si può facilmente vedere che l'implementazione multi-GPU di clMAGMA è meno efficiente di quella intesa per GPU singola: questo è dovuto al maggior numero di trasferimenti effettuati in quest'algoritmo, che appunto trarrebbe un reale vantaggio in termini prestazionali solo se eseguito su almeno due GPU che lavorano in contemporanea. In generale, l'algoritmo in MAGMA è più performante, per via dell'implementazione ottimizzata per hardware NVidia.

### 3.2.2 Sviluppi futuri

Più GPU si usano, migliore sarà la performance, in quanto le latenze dei trasferimenti sarebbero mascherate dalla massiccia attività parallela svolta dalle GPU, sia a livello di accelerazione dei calcoli, sia a livello di computazione simultanea. Pertanto i test dell'algoritmo su un ambiente multi-GPU permetterebbero di valutarne meglio le performance.

Infine, l'algoritmo proposto ha un grado di precisione molto più bassa rispetto all'implementazione in MAGMA, il che suggerisce di approfondire i controlli sui risultati ottenuti dalle singole chiamate alle operazioni di più basso livello, per garantire una miglior accuratezza dei risultati.

## 3.3 Kernel OpenCL cross-platform

Un kernel OpenCL può essere compilato per l'esecuzione su un determinato dispositivo utilizzando le funzioni messe a disposizione dal driver OpenCL di riferimento. Come già accennato, clMAGMA dispone di un sottoprogramma che si occupa della pre-compilazione dei programmi OpenCL per una architettura definita staticamente nel Makefile.

### 3.3.1 Sviluppi futuri

Modificando opportunamente questo sotto-programma, con il supporto dell'inizializzazione dinamica dell'ambiente proposto, è possibile pre-compilare i kernel OpenCL di clMAGMA per diverse architetture, permettendo di sfruttare piattaforme e dispositivi eterogenei, e non solo GPU, per le proprie computazioni

## 3.4 Auto-bilanciamento del carico di lavoro

clMAGMA basa le proprie chiamate BLAS ottimizzate per la GPU sulla libreria clAmdBlas, una libreria scritta in OpenCL che implementa tutte le funzioni BLAS standard, ottimizzata per l'utilizzo su hardware ATI. Questa libreria è compatibile anche per l'esecuzione su GPU NVidia, semplicemente indicando in fase di compilazione i parametri relativi alle caratteristiche computazionali del dispositivo che si vuole utilizzare e la piattaforma da utilizzare per la compilazione dei kernel. clAmdBlas dispone anche di una specifica funzione, chiamata clAmdBlasTune, che permette di ridefinire il ca-

rico di lavoro delle chiamate OpenCL effettuate dalla libreria, in modo da ottimizzare l'esecuzione dei kernel per l'architettura su cui si sta eseguendo.

### 3.4.1 Sviluppi futuri

Utilizzando un meccanismo di ottimizzazione come quello appena descritto, clMAGMA potrebbe rilevare le impostazioni di bilanciamento del workload per ottimizzare l'esecuzione delle funzioni di clAmdBlas e dei kernel specifici della libreria, per ottenere prestazioni globali migliori.

# Conclusioni

La libreria MAGMA si propone di fornire le funzionalità della libreria LAPACK, standard de facto per la computazione legata all'Algebra Lineare, riscritte per il funzionamento sulle nuove architetture ibride composte da GPU e CPU multicore. La possibilità di eseguire il codice solo su GPU NVidia, per quanto efficiente grazie alla maturità di CUDA e del suo ecosistema software, limita tuttavia la possibilità di sfruttare il calcolo parallelo su altri tipi di architetture.

Il porting OpenCL di questa libreria permette una maggiore portabiltià di queste routine e la possibilità di utilizzare ulteriori dispositivi specializzati per le computazioni, aprendo così nuovi scenari d'uso. Tuttavia, per ottenere un'implementazione efficiente di clMAGMA, essa necessita di ulteriori ottimizzazioni e meccanismi che permettano, possibilmente in maniera quasi automatica, di mascherare la difficoltà implementativa allo sviluppatore che voglia utilizzare questa libreria nelle proprie applicazioni.

# Bibliografia

[1] OpenCL 1.2 Manual - Khronos Group Official Website - http://khronos.org

[2] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, S. Tomov, *LU Factorization for Accelerator-based Systems*

[3] A. Buttari, J. Langou, J. Kurzak, J. Dongarra, *A class of parallel tiled linear algebra algorithms for multicore architectures*

[4] E. L. Yip, *Fortran Subroutines for out-of-core solutions of large complex linear systems.* Technical Report CR-159142, NASA, 1979

[5] MAGMA Official Website - http://icl.cs.utk.edu/magma/