Wolfram Mathematica

Il software di riferimento per la Didattica, la Ricerca e lo Sviluppo

WebSeminar Mathematica

Lezione 10

High – Performance Computing e *Mathematica*

Crescenzio Gallo – Università di Foggia

crescenzio.gallo@unifg.it

Note:

- Il materiale visualizzato durante questo seminario è disponibile per il download all'indirizzo http://www.crescenziogallo.it/unifg/seminario-mathematica-2014/
- Per una migliore visione ingrandire lo schermo mediante il pulsante in alto a destra "Schermo intero"

Agenda

Parallelismo

- Introduzione
- Modalità di connessione
- Principali caratteristiche
- Load balancing
- Analisi delle computazioni
- Monitoraggio dei kernel
- Virtual shared memory
- Esempi

GPU programming

- Introduzione
- Considerazioni generali
- Installazione e verifica
- I vantaggi
- Esempi

Compilazione

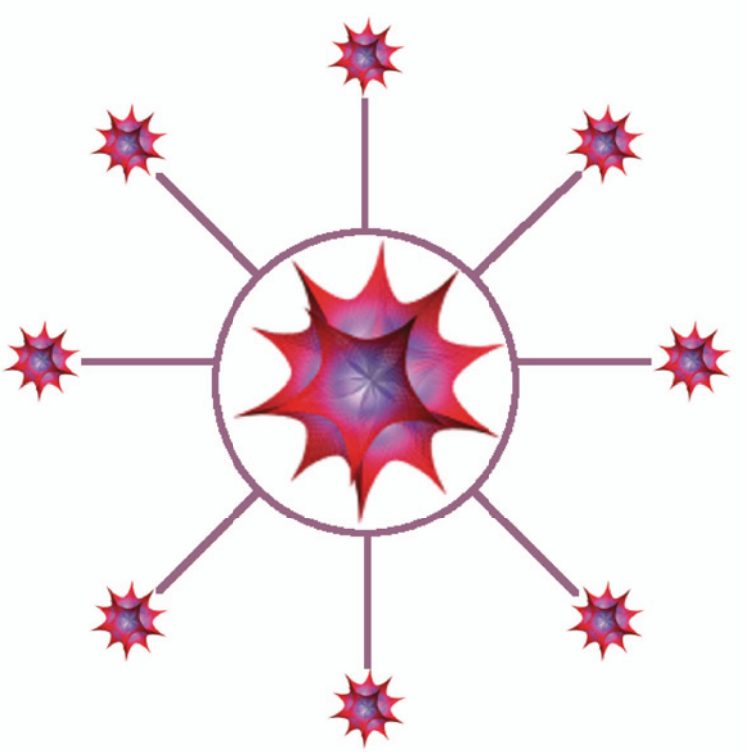
- Introduzione
- Esempi

Conclusioni

◄ | ►

Parallelismo: introduzione

Quando ci sono problemi di tempo di calcolo eccessivo ci sono diverse alternative per migliorare le performance del codice *Mathematica*, tra cui sicuramente una revisione del codice al fine di eliminare eventuali ridondanze o porzioni di codice che possono essere sostituite con comandi nativi (macro) di *Mathematica*.
Comunque, in molte circostanze l'inefficienza non dipende solo dallo stile del codice ma da problemi di complessità computazionale intrinseca al problema trattato o dalle dimensioni dei dati utilizzati. In tali casi ci si può orientare a tecniche quali il parallelismo, la computazione su GPU con i linguaggi CUDA o OpenCL o alla compilazione tramite il comando **Compile**.



Mathematica fornisce un parallellismo di tipo master-slave implementato tramite il protocollo di comunicazione *MathLink*, che consente di collegare un numero elevato di kernel slave ad un kernel master senza richiedere specifiche competenze o sforzi implementativi all'utente.
L'immagine mostra il tipico schema di parallelismo master-slave impiegato da *Mathematica:* il master kernel si preoccupa di analizzare il calcolo richiesto ed automaticamente tentare di scomporlo in sub-task (anche detti job) che possono includere una o più valutazioni a seconda del metodo di parallelismo scelto. I kernel slave (anche detti sub-kernel) non dialogano tra di loro ma sono collegati al master da cui ricevono il job ed a cui restituiscono il risultato. Infine, è sempre il master kernel che aggrega i risultati parziali ottenuti dai singoli sub-kernel e presenta all'utente un unico risultato. Tutto questo è del tutto trasparente per l'utente per cui non è visibile la differenza tra una computazione con un solo kernel e

una con *n* sub-kernel, se non per il fatto che il tempo complessivo di calcolo è notevolmente ridotto.

Parallelismo: modalità di connessione

Kernel locali: per sistemi multi-processore o multi-core. I kernel vengono eseguiti sulla stessa macchina. Questa modalità è l'unica possibile se si possiede una sola licenza di *Mathematica*, che abilita fino a 4 kernel slave più 1 kernel master. Ovviamente, a prescindere dalla licenze, non vengono attivati mai più kernel di quanti processori/core sono disponibili per evitare che due kernel
vadano in esecuzione (sequenzialmente) su uno stesso processore/core.

Lightweight Grid: pacchetto software della Wolfram che serve per gestire la configurazione automatica di kernel paralleli in esecuzione di macchine diverse, dunque per reti eterogenee (sui client bisogna avere le necessarie licenze di *Mathematica*) dove non esiste un software di gestione del tipo "cluster engine".

Cluster integration: *Mathematica* si integra con diversi cluster engine e tecnologie di gestione dei cluster, tra cui:

- Windows Computer Cluster Server
- Windows HPC Server 2008
- Platform™ LSF®
- Altair™ PBS Professional®
- Sun Grid Engine
- Apple Xgrid

Kernel remoti: configurazione manuale di kernel multipli su macchine diverse (sui client bisogna avere le necessarie licenze di *Mathematica*), tramite il Pannello delle Preferenze.

Parallelismo: principali caratteristiche

- Memoria distribuita, parallelismo master/slave
- Indipendenza dal tipo di macchina (grid eterogenee, macchine multiprocessori, LAN, WAN, ecc.)
- Comunicazione basata sul protocollo *MathLink*
- Scambio di espressioni simboliche e programmi e non solo numeri e vettori
- Scheduling dei processi virtuali distribuzione diretta dei processi ai processori disponibili
- Memoria virtuale condivisa, sincronizzazione, locking
- Latency hiding
- Programmazione parallela e supporto al parallelismo automatico
- Failure recovery, riassegnazione automatica dei processi in caso di fallimento di un kernel slave

Parallelismo: load balancing

Ci sono diverse tecniche per suddividere e distribuire un calcolo tra più processori. Le funzioni del tipo **ParallelMap**, **ParallelTable** o **Parallelize** sono dotate di una opzione chiamata **Method** che permette di specificare quale metodo usare, ma in generale lasciando il valore di default **Automatic** si confida sulla capacità del sistema di automatismo di bilanciare, caso per caso, il tempo di comunicazione con il tempo di computazione complessivo. Infatti, in alcuni casi il tempo di coordinamento e comunicazione tra il kernel master e gli slave potrebbe incidere negativamente sul tempo complessivo di calcolo.

Metodo "FinestGrained": vengono creati job individuali per ciascuna computazione elementare. Questa scelta ottimizza il load balancing (alta frammentazione del task ed elevata partecipazione di tutti i kernel) ma a scapito del maggiore tempo di comunicazione richiesto:

```
ParallelMap[Labeled[Framed[#], $KernelID] &, Range[20], Method → "FinestGrained"]
```

```
{[1]8, [2]7, [3]6, [4]5, [5]4, [6]3, [7]2, [8]1, [9]8, [10]7,
 [11]6, [12]5, [13]4, [14]3, [15]2, [16]1, [17]8, [18]7, [19]6, [20]5}
```

Metodo "CoarsedGrained": più valutazioni raggruppate in un singolo job per ciascun kernel. Non si ottimizza il load balancing ma si minimizza il tempo di comunicazione:

```
ParallelMap[Labeled[Framed[#], $KernelID] &, Range[20], Method → "CoarsestGrained"]
```

```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
 1  1  1  2  2  2  3  3  3  4
 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
 4   4   5   5   6   6   7   7   8   8
```

Lasciando al sistema di automazione l'onere del bilanciamento si ha un mix di scelte che cerca di minimizzare il tempo complessivo ed ottimizzare il load balancing (il subkernel più veloce elabora più task):

```
ParallelMap[Labeled[Framed[#], $KernelID] &, Range[20], Method → Automatic]
```

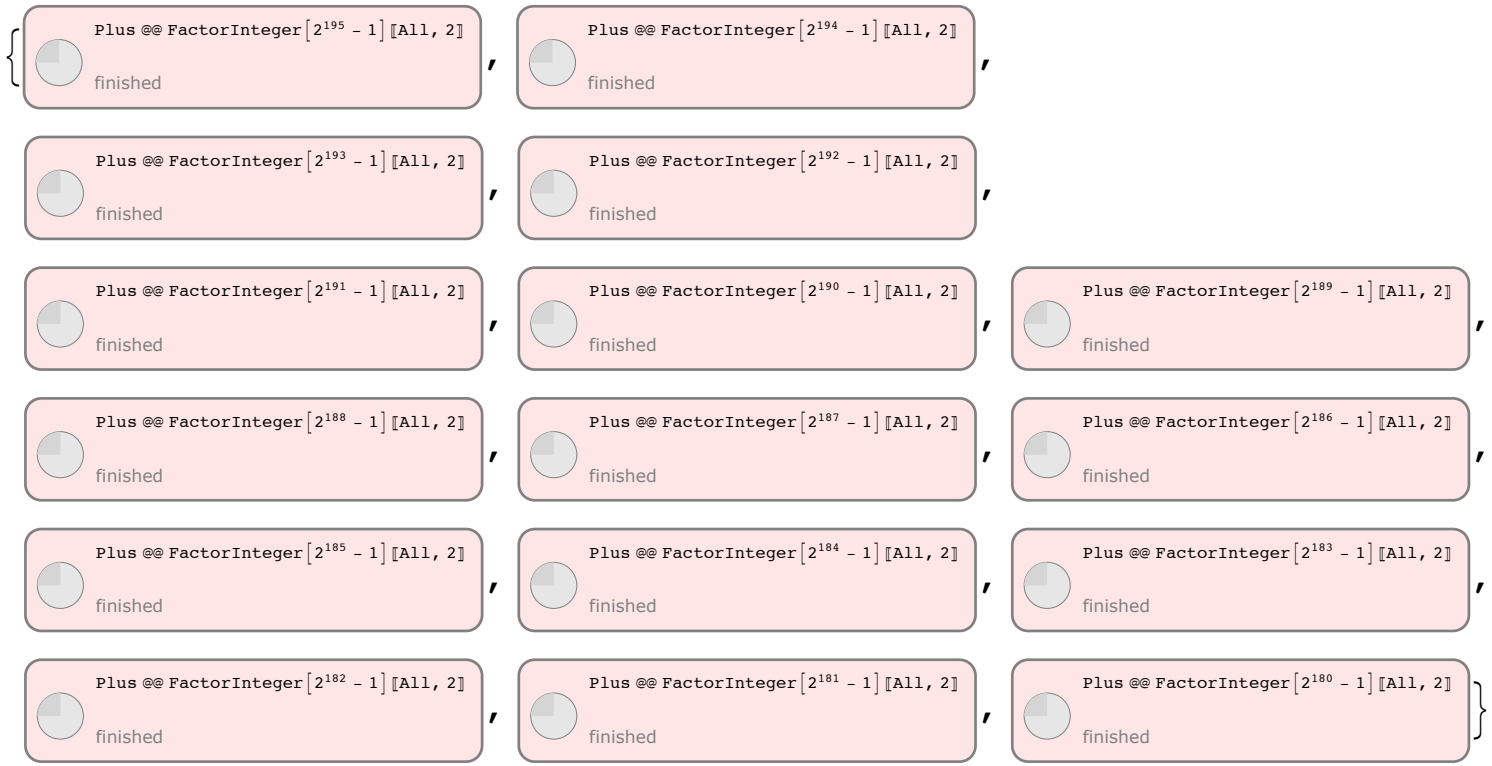
```
{1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
 8  8  7  7  6  6  5  5  4  3
 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
 2   1   8   7   6   5   4   3   2   1
```

Parallelismo: analisi delle computazioni

Ci sono strumenti di visualizzazione dinamica dello scheduling dei task che potrebbero facilitare l'esame delle inefficienze in fase di programmazione

In questo esempio si emula il metodo *FinestGrained* tramite **`ParallelSubmit`**/**`WaitAll`**, inviando tutte le valutazioni individualmente e poi eseguendo lo scheduler e mettendosi in "ascolto" su tutte le valutazioni.

```
fattori =
 Table[ParallelSubmit[{i}, Apply[Plus, FactorInteger[2 ^ i - 1][[All, 2]]]], {i, 195, 180, -1}]
```




```
WaitAll[fattori]
```

```
{8, 7, 3, 16, 5, 10, 10, 10, 5, 8, 5, 11, 5, 11, 4, 24}
```

Oppure è possibile elaborare grafici statistici in real-time, raccogliendo dati durante le elaborazioni per poi fare eventualmente statistiche sui tempi di computazione.

```
Quit[];
```

```
LaunchKernels[];
```

```
?LaunchKernels
```

LaunchKernels[] launches all currently configured parallel subkernels.
LaunchKernels[n] launches n local subkernels on the current computer.
LaunchKernels[des] launches a subkernel with the given description.
LaunchKernels[{des_1, des_2, …}] launches several subkernels with the given descriptions. ≫

Inizializziamo la variabile che accumula i tempi di calcolo dei singoli kernel e condividiamola con tutti i kernel:

```
conta = Table[{0}, {$KernelCount}];
SetSharedVariable[conta];
```

```
?SetSharedVariable
```

SetSharedVariable[s_1, s_2, …] declares the symbols s_i
as shared variables whose values are synchronized among all parallel kernels. ≫

Creiamo un grafico dinamico che si aggiorna man mano che i kernel terminano un job:

```
Dynamic[BarChart[conta,
  PerformanceGoal → "Speed",
  ChartLayout → "Stacked",
  PlotRange → {0, 1},
  ChartLabels → {Placed[{1, 2, 3, 4, 5, 6, 7, 8}, Before], None}]]
```

```
ParallelTable[
 Pause[3];
 Module[{t},
  t = Timing[Plus @@ FactorInteger[2 ^ i - 1][[All, 2]]];
  AppendTo[conta[[$KernelID]], t[[1]]];
  t[[2]]],
 {i, 192, 160, -1}]
```

```
{16, 5, 10, 10, 10, 5, 8, 5, 11, 5, 11, 4, 24, 3,
 5, 6, 14, 9, 11, 4, 10, 7, 7, 4, 19, 2, 8, 10, 10, 5, 16, 7, 13}
```

Parallelismo: monitoraggio dei kernel

Vediamo la differenza in termini di efficienza dei vari metodi nella distribuzione di uno stesso compito su diversi sub-kernel, questa volta con un esempio pratico ed utilizzando il pannello di stato dei sub-kernel.

```
Quit[]
```

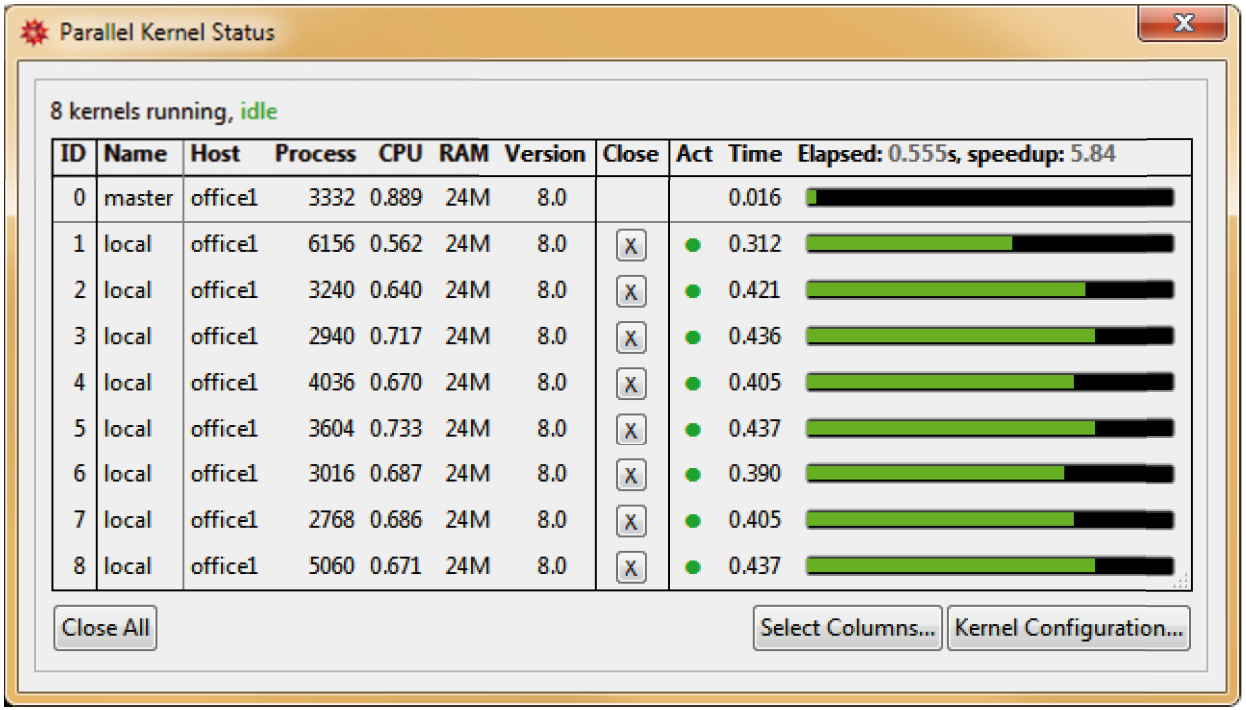
```
LlaunchKernels[];
```

Parallel Kernel Status...

```
AbsoluteTiming[ParallelTable[Integrate[Cos[Sin[p] x], {x, 0.0, 1}],
   {p, 0.0, 1.0, 0.005}, Method → "CoarsestGrained"];]
```

```
{4.738671, Null}
```

Figura 1




```
Quit[]
```

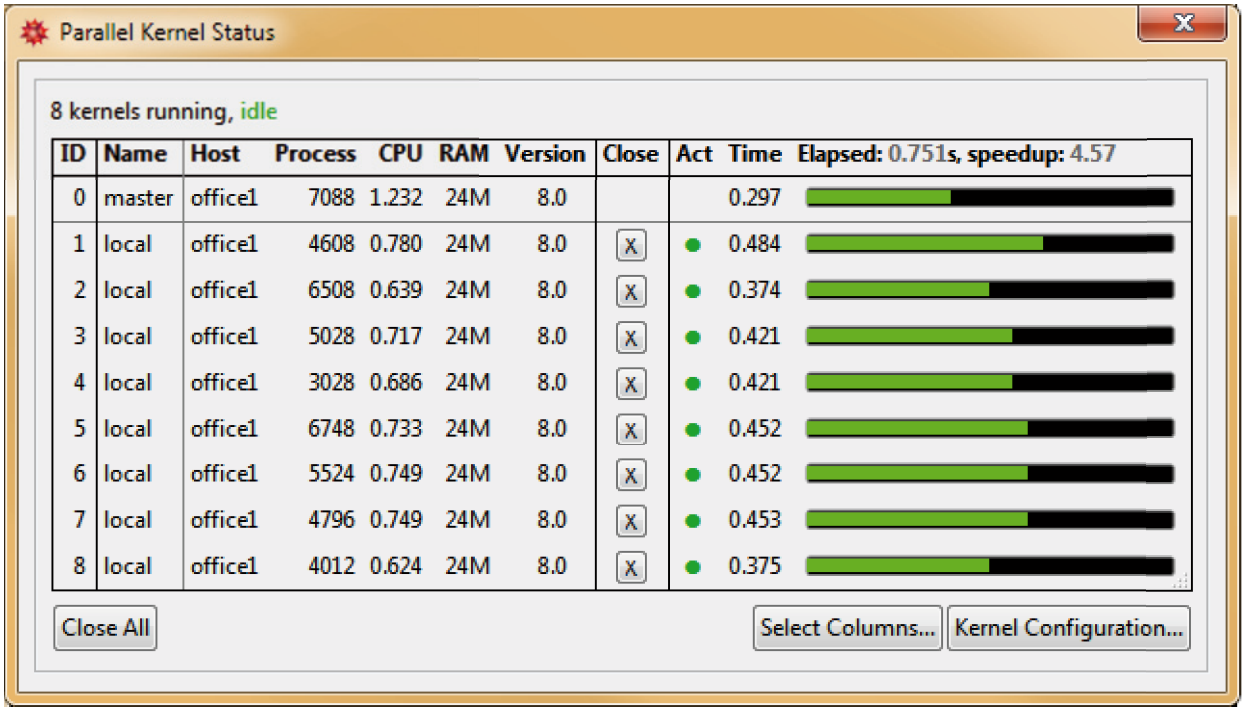
```
LaunchKernels[];
```

Parallel Kernel Status...

```
AbsoluteTiming[ParallelTable[Integrate[Cos[Sin[p] x], {x, 0.0, 1}],
   {p, 0.0, 1.0, 0.005}, Method → "FinestGrained"];]
```

```
{0.6240011, Null}
```

Figura 2




```
Quit[]
```

```
LaunchKernels[];
```

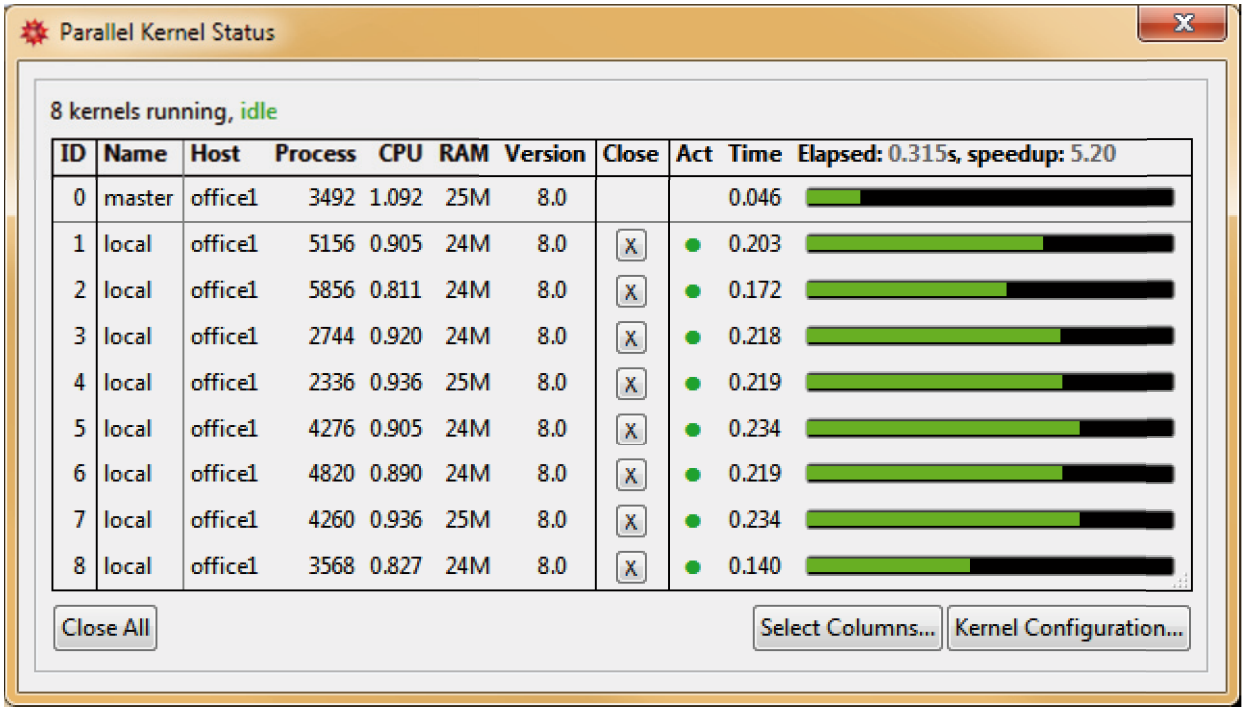
Parallel Kernel Status...

Come visto prima il metodo **Automatic** (default) cerca il migliore equilibrio:

```
AbsoluteTiming[ParallelTable[Integrate[Cos[Sin[p] x], {x, 0.0, 1}], {p, 0.0, 1.0, 0.005}];]
```

```
{2.931844, Null}
```

Figura 3


Parallel Kernel Status
8 kernels running, idle
ID Name Host Process CPU RAM Version Close Act Time Elapsed: 0.315s, speedup: 5.20
0 master office1 3492 1.092 25M 8.0 0.046
1 local office1 5156 0.905 24M 8.0 X 0.203
2 local office1 5856 0.811 24M 8.0 X 0.172
3 local office1 2744 0.920 24M 8.0 X 0.218
4 local office1 2336 0.936 25M 8.0 X 0.219
5 local office1 4276 0.905 24M 8.0 X 0.234
6 local office1 4820 0.890 24M 8.0 X 0.219
7 local office1 4260 0.936 25M 8.0 X 0.234
8 local office1 3568 0.827 24M 8.0 X 0.140
Close All
Select Columns...
Kernel Configuration...

Parallelismo: virtual shared memory

L'architettura del parallelismo di *Mathematica* è evidentemente a memoria distribuita in quanto i kernel slave hanno una gestione indipendente della memoria loro allocata, anche se si trovano sulla stessa macchina dove esegue anche il master. Però, poichè in molti casi è indispensabile condividere le stesse variabili, viene implementato un meccanismo di memoria condivisa virtuale che permette di gestire variabili condivise tra i kernel slave. Di fatto le variabili che vengono definite "shared" vengono mantenute in memoria dal kernel master e tutti gli slave vi possono accedere. Chiaramente questo meccanismo, non essendo basato su una vera e propria memoria fisica condivisa (appunto "virtual shared") incrementa il tempo di comunicazione tra i kernel slave ed il master.

Per ragioni tecniche le variabili condivise devono sempre avere un valore assegnato, pertanto è conveniente dichiararle ed assegnarvi un valore prima di rendere "shared":

```
x = 17;
```

```
SetSharedVariable[x]
```

```
r1 = Kernels[][[1]];
r2 = Kernels[][[2]];
```

```
?Kernels
```

Kernels[] gives the list of running kernels available for parallel computing. ≫

```
ParallelEvaluate[x, r1]
```

```
17
```

```
ParallelEvaluate[x = 18, r2]
```

```
18
```

```
x
```

```
18
```

```
ParallelEvaluate[x, r1]
```

```
18
```

```
ParallelEvaluate[x]
```

```
{18, 18, 18, 18, 18, 18, 18, 18}
```

```
? ParallelEvaluate
```

ParallelEvaluate[*expr*] evaluates the expression
expr on all available parallel kernels and returns the list of results obtained.
ParallelEvaluate[*expr*, *kernel*] evaluates *expr* on the parallel kernel specified.
ParallelEvaluate[*expr*, {ker_1, ker_2, …}] evaluates *expr* on the parallel kernels ker_i. ≫

Parallelismo: virtual shared memory

L'uso di variabili condivise introduce anche il problema della sincronizzazione. Eventuali accessi continui a variabili condivise potrebbero sovrapporsi, e potrebbe capitare che un mentre un kernel sta salvando un dato in una variabile condivisa un altro kernel accede alla stessa variabile, prendendo così il valore precedente.

In questo esempio usiamo la funzione **Pause** per aumentare la probabilità di sovrapposizione tra due operazione di lettura/scrittura della stessa variabile condivisa.

```
Quit[]
```

Proviamo il calcolo in sequenziale per verificare il risultato corretto:

```
a = 0;
Map[(a = a + 1) &, Range[10]];
a
```

```
10
```

Effettuando lo stesso calcolo in parallelo, si sovrappongono diversi accessi read/write:

```
b = 0;
SetSharedVariable[b];
ParallelMap[(b = b + 1) &, Range[10]];
b
```

```
2
```

In casi come questi si può bloccare l'accesso alla variabile condivisa tramite la funzione **CriticalSection** sincronizzando così gli accessi read/write:

```
Quit[]
```

```
b = 0;
SetSharedVariable[b];
ParallelMap[CriticalSection[{lock}, (b = b + 1)] &, Range[10]];
b
```

```
10
```

```
?CriticalSection
```

CriticalSection[{var_1, var_2, …}, *expr*] locks the variables
var_i with respect to parallel computation, evaluates *expr*, then releases the var_i. ≫

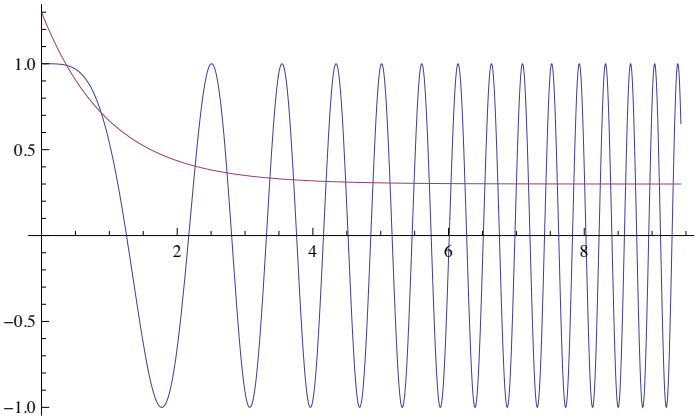
Nota: **CriticalSection** deve avere come variabile locale *(lock* nell'esempio) una variabile non definita nel master kernel perchè in questa variabile verrà memorizzato l'ID del kernel in esecuzione e questo serve per "congelare" gli accessi alle variabili presenti nel body del **CriticalSection** stesso.

Parallelismo: esempi

Calcolo di radici di un'equazione.

Vogliamo calcolare le radici dell'equazione $\cos(t^2) = e^{-t} + 0.3$ utilizzando il metodo di Newton. La funzione FindRoot risulta utile in questo esempio perchè permette di specificare un punto iniziale da cui far partire l'algoritmo nella ricerca di una soluzione. Utilizzando più punti di partenza possiamo trovare tutte le soluzioni in un dato intervallo, e questo procedimento può trarre ovvio giovamento dall'uso di una tecnica di parallellizzazione.

```
plot = Plot[{Cos[t^2], e^-t + .3}, {t, 0, 3 π}]
```



La variabile "start" contiene dei possibili valori iniziali per la ricerca delle soluzioni locali (ovviamente in funzione della granularità dei punti iniziali sull'intervallo preso si possono soluzioni calcolate più di una volta da più istanze della **FindRoot**, dunque alla fine la lunghezza del vettore soluzioni è maggiore del numero delle soluzioni stesse essendoci alcune ripetizioni).

```
start = Table[x, {x, 0, 3 π, .1}];
```

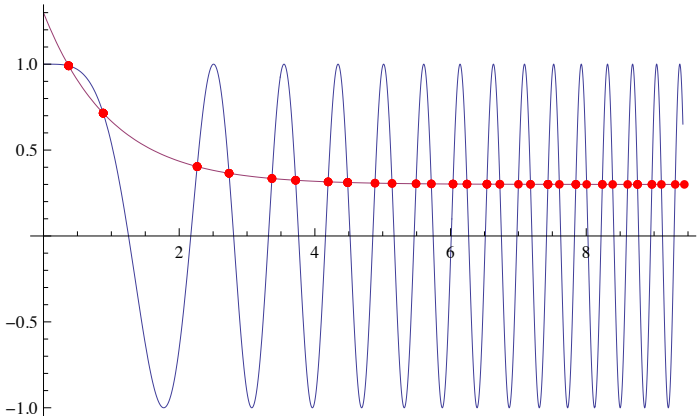
Ora si calcolano le soluzioni mappando la funzione **FindRoot** su ciascun punto iniziale definito nella variabile “start”:

```
ris = Map[FindRoot[{Cos[t^2] == Exp[-t] + .3}, {t, #1}] &, start]; // AbsoluteTiming
```

```
{0.063206, Null}
```

Verifichiamo la bontà delle soluzioni con un grafico congiunto:

```
Show[plot, Graphics[
  {Red, PointSize[Medium], Point[MapThread[{t /. #1, #2} &, {ris, Cos[t^2 /. ris]}]]}]]
```



Rifacciamo adesso lo stesso calcolo utilizzando però la **ParallelMap** al posto della **Map**.

```
LaunchKernels[];
```

```
risp = ParallelMap[FindRoot[{Cos[t^2] == Exp[-t] + .3}, {t, #1}] &, start]; // AbsoluteTiming
```

```
{0.028528, Null}
```

```
ris == risp
```

```
True
```

Proviamo ad incrementare la dimensione dell'intervallo (e di conseguenza il numero di soluzioni da trovare) e ad aumentare la precisione di calcolo richiesta, tanto per avere un'idea più consistente del vantaggio del calcolo parallelo.

```
start = Table[x, {x, 0, 30 π, .1`50}];
```

```
ris = Map[FindRoot[{Cos[t^2] == Exp[-t] + .3`50}, {t, #1}, WorkingPrecision → 50] &, start]; //
 AbsoluteTiming
```

```
{1.550749, Null}
```

```
risp = ParallelMap[FindRoot[{Cos[t^2] == Exp[-t] + .3`50}, {t, #1}, WorkingPrecision → 50] &,
    start]; // AbsoluteTiming
```

```
{0.286945, Null}
```

```
ris == risp
```

```
True
```

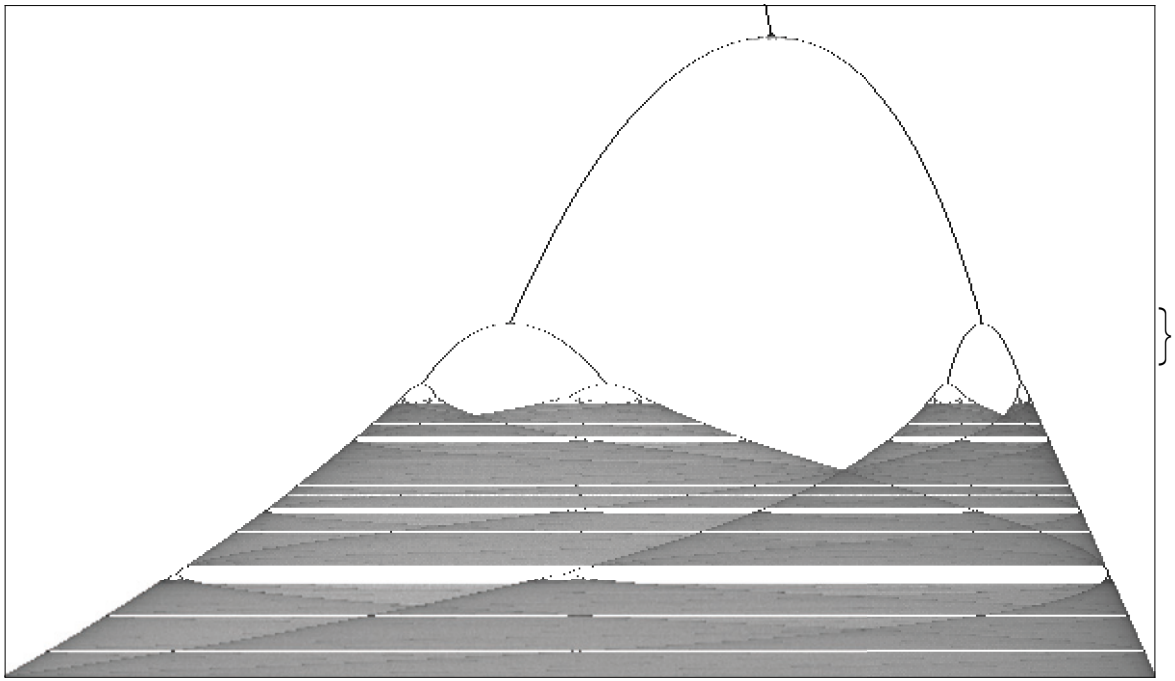
Parallelismo: esempi

Calcolare e visualizzare un diagramma Feigenbaum (o biforcazione).

```
line[r_, dy_, np_, n0_, n_] := Module[{pts},
  With[{logistics = Function[x, r x (1 - x)]},
   pts = Join @@ NestList[logistics, Nest[logistics, RandomReal[{0, 1}, np], n0], n - 1]];
  Log[1.0 + BinCounts[pts, {0, 1, dy}]]]
```

```
AbsoluteTiming[
 With[{w = 600, h = 350, r0 = 2.95, r1 = 4.0},
  ArrayPlot[ParallelTable[line[r, 1 / (w - 1), w, 1000, 100], {r, r0, r1, (r1 - r0) / (h - 1)}],
   ImageSize → {w, h}, PixelConstrained → True]]]
```

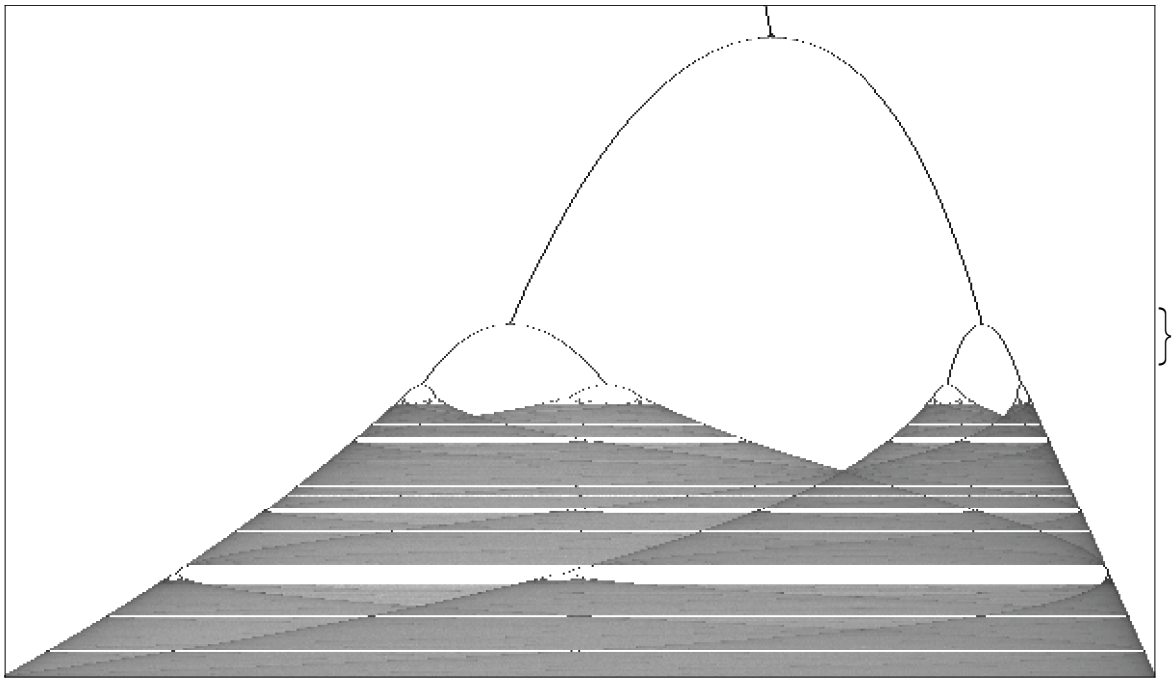
```
{1.326997,
```



```
}
```

```
AbsoluteTiming[With[{w = 600, h = 350, r0 = 2.95, r1 = 4.0},
  ArrayPlot[Table[line[r, 1 / (w - 1), w, 1000, 100], {r, r0, r1, (r1 - r0) / (h - 1)}],
   ImageSize → {w, h}, PixelConstrained → True]]]
```

```
{5.485587,
```

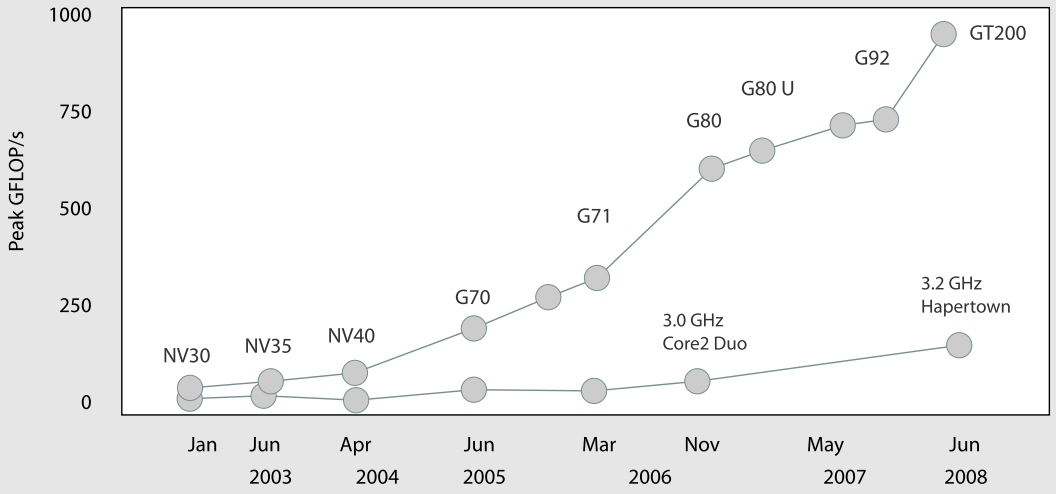


```
}
```

GPU programming: introduzione

L'evoluzione delle schede grafiche, soprattutto per motivi commerciali legati alla enorme diffusione di giochi con interfacce grafiche evolute e molto esose dal punto di vista dei calcoli numerici real-time, ha portato allo sviluppo di nuove architetture hardware per il calcolo numerico veloce con un livello di parallelismo elevato. Insieme a tali nuovi modelli di schede, all'incira negli anni 2006/2007 sono anche nati nuovi linguaggi, quali il CUDA e poi OpenCL, che servono proprio a programmare routine numeriche da eseguire sui core computazionali in dotazione alle schede grafiche, da cui il termine GPU che sta per *Graphical Processing Unit*.

L'immagine mostra una comparazione di performance tra GPU e CPU:



GT200 - GeForce GTX200	G71 - GeForce 7900 GTX	NV35 - GeForce FX 5950 U
G92 - GeForce 9800 GTX	G70 - GeForce 7800 GTX	NV30 - GeForce FX 5800
G80 - GeForce 8800 GTX	NV40 - GeForce 6800 Ultra	

GPU programming: introduzione

Grazie ad alcune tecnologie di base aggiunte in *Mathematica* 8 è stato possibile sviluppare ed includere anche i package CUDALink e OpenCLLink. Infatti, questi due package sono basati su un uso intensivo di

- ***LibraryLink*** per caricare librerie C in *Mathematica*
- **SymbolicC** per generare codice C in *Mathematica*
- **Compiler** per velocizzare le computazioni su GPU in *Mathematica*

◄ | ►

GPU programming: considerazioni generali

Vediamo alcuni caratteri distintivi dei due linguaggi.

CUDA

- Linguaggio di programmazione sviluppato dalla NVIDIA verso la fine del 2007
- Linguaggio vicino al C/C++

```
__global__ void addTwo_kernel(int * arry, len) {
        int index = threadIdx.x + blockIdx * blockDim.x;
        if (index >= len) return;
        arry[index] += 2;
}
```

- Considera la GPU come una generica unità di elaborazione
- Molto diffuso, migliaia di sviluppatori in tutto il mondo

VANTAGGI:

> Supporta caratteristiche simili al C++ come ad esempio i template
> Linguaggio più maturo per il calcolo su GPU
> Consente la compilazione offline senza pregiudicare le performance

OpenCL

- Linguaggio di programmazione sviluppato da Khronos verso la fine del 2008
- Standard *open* supportato da Apple, NVIDIA, e AMD/ATI
- Linguaggio vicino al C

```
__kernel void addTwo_kernel(__global int * arry, len) {
        int index = get_global_id(0);
        if (index >= len) return;
```

```
        arry[index] += 2;
    }
```

- Considera la GPU come una generica unità di elaborazione
- Relativamente nuovo, la prima implementazione stabile risale al 2009

VANTAGGI:

> Supporta caratteristiche simili al C ma non i template
> Standard *open*
> Ha un file di specifiche

GPU programming: installazione e verifica

Per un corretto funzionamento dei package CUDA e OpenCL bisogna installare propriamente i driver dell'hardware ed il package di *Mathematica*. Essendo questi dei passaggi delicati, il Documentation Center offre una serie di tutorial che guidano alla corretta configurazione ed uso, se necessario, dei tool di verifica e correzione errori di setup.
I principali tutorial sono *CUDALink* User Guide e Core Setup and Testing.

Vediamo ora alcune funzioni elementari che servono a testare se *Mathematica* può procedere con l'uso dei pacchetti di programmazione GPU.
Innanzitutto, bisogna sempre ricordarsi che sia CUDALink sia OpenCLLink sono pacchetti che non vengono caricati all'avvio di *Mathematica* ma dobbiamo esplicitamente caricarli prima di usare le loro funzioni:

```
Needs["CUDALink`"]
```

```
Needs["OpenCLLink`"]
```

Ora per verificare se *Mathematica* riconosce correttamente i due pacchetti possiamo usare i test:

```
CUDAQ[]
```
```
False
```

```
OpenCLQ[]
```
```
True
```

A questo punto possiamo anche verificare la versione ed altre info con i seguenti:

```
CUDAInformation[]
```
```
CUDAInformation::invdriv :
  CUDA was not able to find a valid CUDA driver. Refer to CUDALink System Requirements for system requirements. ≫
```

```
CUDAInformation[]
```

```
OpenCLInformation[]
```

```
{1 → {Version → OpenCL 1.2 (Dec  8 2013 21:07:05), Name → Apple, Vendor → Apple,
    Extensions → {cl_APPLE_SetMemObjectDestructor, cl_APPLE_ContextLoggingFunctions,
      cl_APPLE_clut, cl_APPLE_query_kernel_names, cl_APPLE_gl_sharing, cl_khr_gl_event},
    1 → {Type → CPU, Name → Intel(R) Xeon(R) CPU           X5472  @ 3.00GHz,
      Version → OpenCL 1.2, Extensions →
       {cl_APPLE_SetMemObjectDestructor, cl_APPLE_ContextLoggingFunctions, cl_APPLE_clut,
        cl_APPLE_query_kernel_names, cl_APPLE_gl_sharing, cl_khr_gl_event, cl_khr_fp64,
        cl_khr_global_int32_base_atomics, cl_khr_global_int32_extended_atomics,
        cl_khr_local_int32_base_atomics, cl_khr_local_int32_extended_atomics,
        cl_khr_byte_addressable_store, cl_khr_int64_base_atomics,
        cl_khr_int64_extended_atomics, cl_khr_3d_image_writes, cl_khr_image2d_from_buffer,
        cl_APPLE_fp64_basic_ops, cl_APPLE_fixed_alpha_channel_orders,
        cl_APPLE_biased_fixed_point_image_formats, cl_APPLE_command_queue_priority},
      Driver Version → 1.1, Vendor → Intel, Profile → FULL_PROFILE,
      Vendor ID → 4 294 967 295, Compute Units → 8, Core Count → 8,
      Maximum Work Item Dimensions → 3, Maximum Work Item Sizes → {1024, 1, 1},
      Maximum Work Group Size → 1024, Preferred Vector Width Character → 16,
      Preferred Vector Width Short → 8, Preferred Vector Width Integer → 4,
      Preferred Vector Width Long → 2, Preferred Vector Width Float → 4,
      Preferred Vector Width Double → 2, Maximum Clock Frequency → 3000, Address Bits → 64,
      Maximum Memory Allocation Size → 3 221 225 472, Image Support → True,
      Maximum Read Image Arguments → 128, Maximum Write Image Arguments → 8,
      Maximum Image2D Width → 8192, Maximum Image2D Height → 8192,
      Maximum Image3D Width → 2048, Maximum Image3D Height → 2048,
      Maximum Image3D Depth → 2048, Maximum Samplers → 16, Maximum Parameter Size → 4096,
```

```
Memory Base Address Align → 1024, Memory Data Type Align Size → 128,
Floating Point Precision Configuration → {Denorms, Infinity, NaNs,
  Round to Nearest, Round to Infinity, Round to Zero, IEEE754-2008 Fused MAD},
Global Memory Cache Type → Read Write, Global Memory Cache Line Size → 6 291 456,
Global Memory Cache Size → 64, Global Memory Size → 12 884 901 888,
Maximum Constant Buffer Size → 65 536, Maximum Constant Arguments → 8,
Local Memory Type → Global, Local Memory Size → 32 768,
Error Correction Support → False, Profiling Timer Resolution → 1,
Endian Little → True, Available → True, Compiler Available → True,
Execution Capabilities → {Kernel Execution, Native Kernel Execution},
Command Queue Properties → {Profiling Enabled}}}}
```

Infine, un modo sintetico per verificare l'intero sistema è **SystemInformation[]**, una particolare utility di *Mathematica* che fornisce un riepilogo completo sul sistema in uso:

```
InstallJava[]
```

```
LinkObject['/Applications/Mathematica
   9-0-1-0.app/SystemFiles/Links/JLink/JLink.app/Contents/MacOS/JavaApplicationStub'
   -init "/tmp/m000003361431", 9726, 15]
```

```
SystemInformation[]
```

Kernel | Front End | Links | Parallel | Devices | Network

Version	9.0 for Mac OS X x86 (64-bit) (January 24, 2013)
Release ID	9.0.1.0 (4055646, 4055073)
Patch Level	0
Activation Key	4718-0042-TQPWHR
Machine ID	5120-02601-13704
User Name	cgallo
Machine Name	server1
Machine Domains	{}
License Server	server1.dmcs.unifg.it
Max License Processes	2
Machine Type	PC
Operating System	MacOSX
Processor Type	x86-64
Language	English
Character Encoding	UTF-8
System Character Encoding	UTF-8
Time Zone	1.
Creation Date	Thu 24 Jan 2013 20:31:41
Installation Directory	/Applications/Mathematica 9-0-1-0.app ≫
Initialization Files Loaded	▶ *4 files*

▶ **Directories**

▶ **Packages & Files**

▶ **Streams**

▶ **MathLink**

▶ **External Compilers**

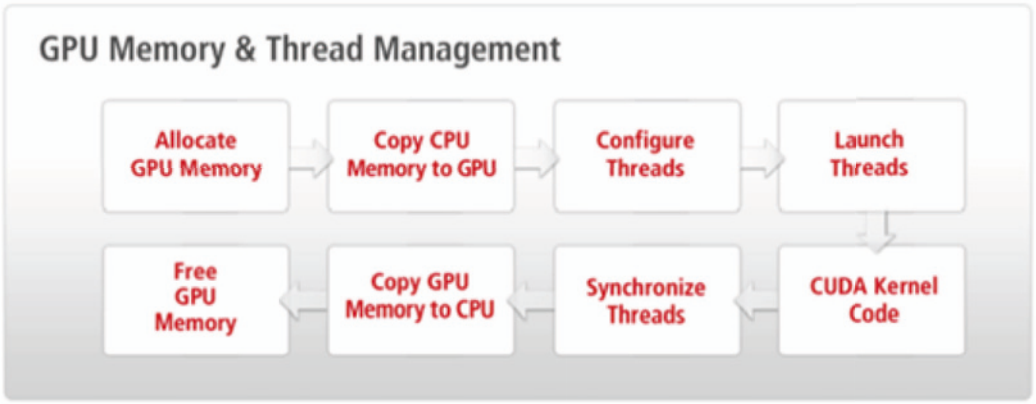
▶ **Advanced**

Benchmark with *MathematicaMark* ...

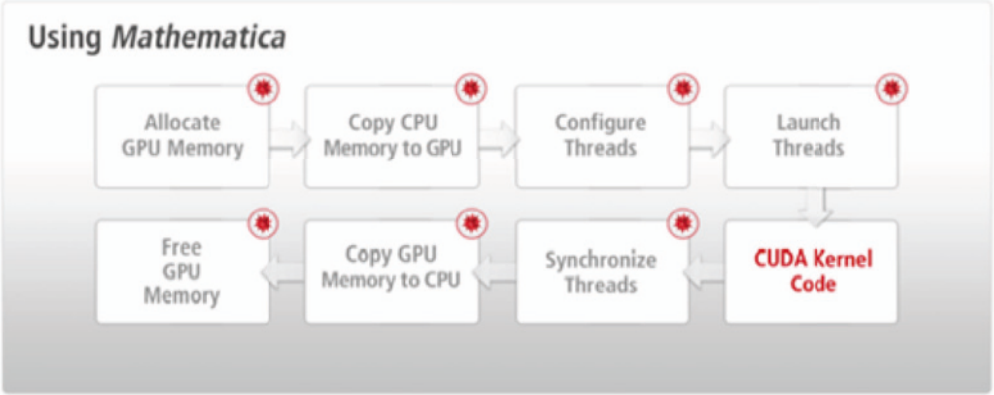
Copy

GPU programming: i vantaggi

Flusso di lavoro semplificato: la programmazione GPU richiede all'utente di occuparsi anche degli step relativi alla compilazione, getione della memoria e dei threads.



I pacchetti *CUDALink* e *OpenCLLink* semplificano il workflow automatizzando la maggior parte di tali step e lasciando all'utente il solo compito di programmare la parte "algoritmica" della funzione CUDA.



Ulteriori vantaggi si possono sintetizzare in:

- Integrazione con le potenzialità native (ed uniche) di *Mathematica*

I nostri programmi possono ovviamente anche essere di tipo ibrido, ossia in parte fare affidamento sulle caratteristiche native di *Mathematica*, ad esempio il calcolo simbolico, il parallelismo su CPU, ecc. ed in parte fare affidamento su porzioni di codice CUDA.

- Ready-to-use applications

L'integrazione di CUDA in *Mathematica* è anche offerta tramite una serie di funzioni native che sono implementate direttamente su linguaggio CUDA, pertanto sfruttano al massimo la tecnologia GPU senza che l'utente debba conoscerne i linguaggi.

- Zero device configuration

Mathematica individua e configura automaticamente le periferiche hardware compatibili con CUDA e OpenCL.

- Supporto per multiple GPU

Grazie al parallelismo di *Mathematica* è possibile anche invocare funzioni CUDA/OpenCL da più kernel contemporaneamente, oppure se il computer dispone di più dispositivi GPU si possono anche impostare programmi CUDA/OpenCL che utilizzano contemporaneamente più schede grafiche.

GPU programming: esempi

Come detto precedentemente *Mathematica* dispone di una serie di funzioni native CUDA, vediamone qualche esempio.

```
Needs["CUDALink`"]
```

```
CUDAImageMultiply[ , ]
```



```
ImageMultiply[ , ]
```


```
Manipulate[CUDAImageAdd[CUDAImageMultiply[ , 1 - alpha],
```

```
  CUDAImageMultiply[ , alpha]], {alpha, 0.0, 1.0, .001}]
```

alpha

```
CUDAImageAdd[CUDAImageMultiply[ , 1.], CUDAImageMultiply[ , 0.]]
```

```
v = Range[1., 10];
CUDAFourier[v]
```

```
{17.39 + 0. i, -1.581 - 4.866 i, -1.581 - 2.176 i, -1.581 - 1.149 i, -1.581 - 0.5137 i,
 -1.581 + 0. i, -1.581 + 0.5137 i, -1.581 + 1.149 i, -1.581 + 2.176 i, -1.581 + 4.866 i}
```

```
v = Range[1., 100];
CUDAFourier[v]
```

```
{505. + 0. i, -5. - 159.1 i, -5. - 79.47 i, -5. - 52.89 i, -5. - 39.58 i, -5. - 31.57 i,
 -5. - 26.21 i, -5. - 22.37 i, -5. - 19.47 i, -5. - 17.21 i, -5. - 15.39 i, -5. - 13.89 i,
 -5. - 12.63 i, -5. - 11.55 i, -5. - 10.63 i, -5. - 9.813 i, -5. - 9.095 i, -5. - 8.455 i,
 -5. - 7.879 i, -5. - 7.357 i, -5. - 6.882 i, -5. - 6.446 i, -5. - 6.044 i, -5. - 5.671 i,
 -5. - 5.324 i, -5. - 5. i, -5. - 4.695 i, -5. - 4.408 i, -5. - 4.136 i, -5. - 3.878 i,
 -5. - 3.633 i, -5. - 3.398 i, -5. - 3.173 i, -5. - 2.957 i, -5. - 2.749 i, -5. - 2.548 i,
 -5. - 2.353 i, -5. - 2.164 i, -5. - 1.98 i, -5. - 1.8 i, -5. - 1.625 i, -5. - 1.453 i,
 -5. - 1.284 i, -5. - 1.118 i, -5. - 0.9538 i, -5. - 0.7919 i, -5. - 0.6316 i,
 -5. - 0.4726 i, -5. - 0.3146 i, -5. - 0.1572 i, -5. + 0. i, -5. + 0.1571 i, -5. + 0.3146 i,
 -5. + 0.4726 i, -5. + 0.6316 i, -5. + 0.7919 i, -5. + 0.9538 i, -5. + 1.118 i,
 -5. + 1.284 i, -5. + 1.453 i, -5. + 1.625 i, -5. + 1.8 i, -5. + 1.98 i, -5. + 2.164 i,
 -5. + 2.353 i, -5. + 2.548 i, -5. + 2.749 i, -5. + 2.957 i, -5. + 3.173 i, -5. + 3.398 i,
 -5. + 3.633 i, -5. + 3.878 i, -5. + 4.136 i, -5. + 4.408 i, -5. + 4.695 i, -5. + 5. i,
 -5. + 5.324 i, -5. + 5.671 i, -5. + 6.044 i, -5. + 6.446 i, -5. + 6.882 i, -5. + 7.357 i,
 -5. + 7.879 i, -5. + 8.455 i, -5. + 9.095 i, -5. + 9.813 i, -5. + 10.63 i, -5. + 11.55 i,
 -5. + 12.63 i, -5. + 13.89 i, -5. + 15.39 i, -5. + 17.21 i, -5. + 19.47 i, -5. + 22.37 i,
 -5. + 26.21 i, -5. + 31.57 i, -5. + 39.58 i, -5. + 52.89 i, -5. + 79.47 i, -5. + 159.1 i}
```

```
v = Range[1., 100 000];
AbsoluteTiming[CUDAFourier[v];]
```

```
{0.0156000, Null}
```

```
v = Range[1., 100 000];
AbsoluteTiming[Fourier[v];]
```

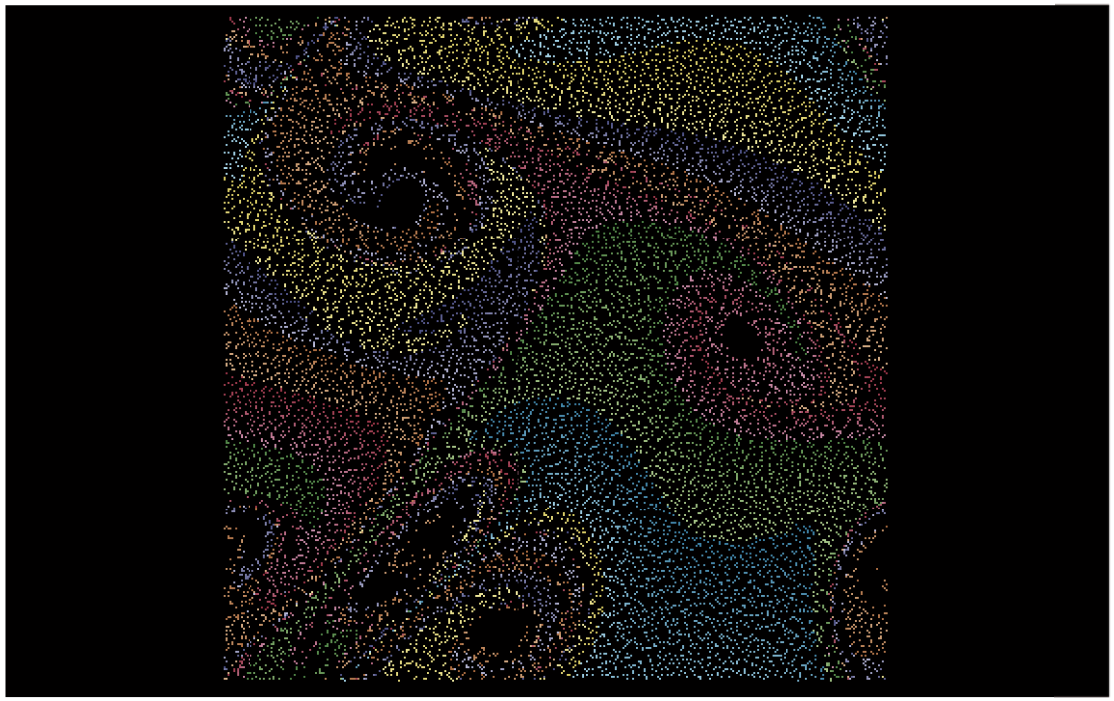
```
{0.0312001, Null}
```

Altri esempi dal Documentation Center Examples

```
Quit[]
```

```
Needs["CUDALink`"];
```

```
CUDAFluidDynamics[]
```



Options

```
Quit[]
```

```
Needs["CUDALink`"]
```

Moltiplicazione di matrici

```
A = RandomInteger[1, {1500, 1500}];
B = RandomInteger[1, {1500, 1500}];
```

```
AbsoluteTiming[res = CUDADot[A, B];]
```

```
{2.1216038, Null}
```

```
AbsoluteTiming[res1 = Dot[A, B];]
```

```
{5.2260092, Null}
```

GPU programming: esempi

Esempio di funzione con OpenCL creata dall'utente: modificare un'immagine facendo la negazione del colore.

```
Quit[]
```

```
Needs["OpenCLLink`"]
```

```
src = "
__kernel void imageColorNegate(__global
    mint * img, mint width, mint height, mint channels) {
    mint ii;
    int xIndex = get_global_id(0);
    int yIndex = get_global_id(1);
    int index = channels*(xIndex + yIndex*width);
    if (xIndex < width && yIndex < height) {
        for (ii = 0; ii < channels; ii++)
            img[index+ii] = 255 - img[index+ii];
    }
}";
```

```
negaOCL = OpenCLFunctionLoad[src, "imageColorNegate",
  {{_Integer}, _Integer, _Integer, _Integer}, {16, 16}, "Defines" → {"mint" → "int"}]
```

```
OpenCLFunction[<>, imageColorNegate, {{_Integer}, Integer64, Integer64, Integer64}]
```

```
img =  ;
{width, height, channels} = Flatten[{ImageDimensions[img], ImageChannels[img]}];
negaOCL[img, width, height, channels]
```

```
OpenCLFunction[<>, imageColorNegate, {{_Integer}, Integer64, Integer64, Integer64}][
  , 168, 168, 3]
```

Costruiamo una Manipulate che applica la funzione a più immagini.

```
imgs = {  ,  ,  ,  };
```

```
Manipulate[
 {width, height, channels} = Flatten[{ImageDimensions[img], ImageChannels[img]}];
 First[negaOCL[img, width, height, channels]],
 {img, imgs}
]
```

Compilazione: introduzione

In *Mathematica* esiste da anni un comando che si chiama **Compile** che consente di compilare funzioni dell'utente che includono calcolo numerico. Con la versione 8 di *Mathematica* questa componente è stata fortemente rinnovata, aggiungendo una serie di opzioni e nuove funzionalità per cui ora un altro modo di velocizzare i propri calcoli può essere proprio suggerito dalla compilazione del codice utente.
Sebbene non in maniera diretta, e senza che noi ce ne accorgiamo, molte volte *Mathematica* utilizza la funzione **Compile** per un'ampia gamma di calcoli: nel plotting, nelle computazioni numeriche del tipo **NDSolve**, **NIntegrate**, **FindRoot**, ecc.
Dunque, la compilazione è di fatto una componente importante del sistema *Mathematica* e contribuisce all'ottimizzazione delle performance.

Da *Mathematica* è possibile

- **Generare** codice C
- **Compilare** codice C code per librerie dinamiche o eseguibili standalone
- **Collegare** librerie dinamiche
- Automaticamente fare tutto quanto sopra detto tramite la funzione **Compile** ed eseguire le funzioni compilate in parallelo

Le funzioni compilate con **Compile** si comportano esattamente come altre funzioni di *Mathematica* solo che sono, in genere, più veloci.

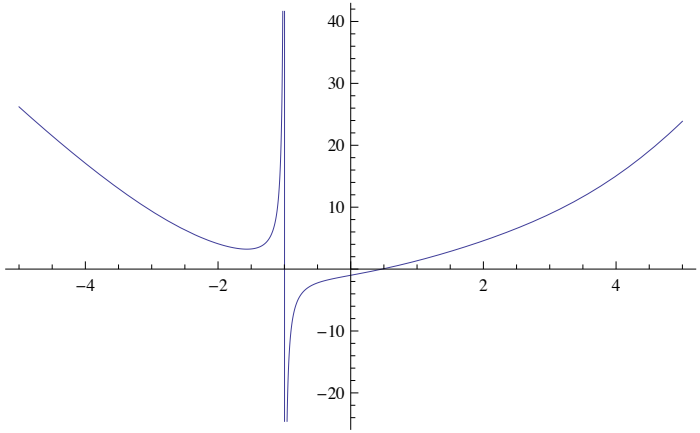
```
cf = Compile[{{x, _Real}}, Sin[x] + x^2 - 1/(1 + x)]
```

$$\text{CompiledFunction}\left[\{x\}, \text{Sin}[x] + x^2 - \frac{1}{1+x}, -\text{CompiledCode}-\right]$$

```
cf[1.2]
```

1.917

```
Plot[cf[x], {x, -5, 5}]
```



Compilazione: introduzione

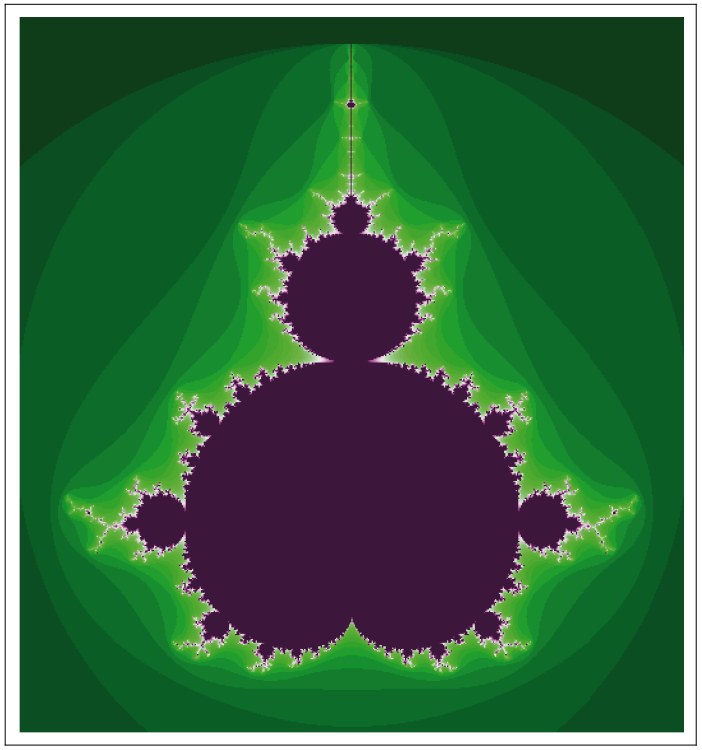
Oltre alla funzione **Compile** in sé, vi sono numerose nuove funzionalità che permettono la generazione di codice C, il link di librerie dinamiche e molto altro ancora. Si suggerisce did are uno sguardo ai seguenti documenti:
The *Mathematica* Compiler User Guide
CCompilerDriver User Guide
CCompilerDriver
C/C++ Language Interface

Compilazione: esempi

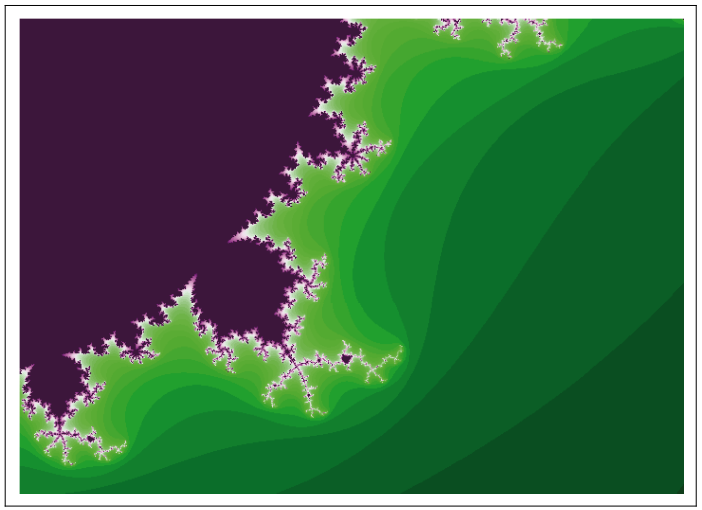
Una funzione per la visualizzazione di un frattale.

```
f = Compile[{{c, _Complex}}, Module[{i = 1}, FixedPoint[(i++; #^2 + c) &, 0, 50,
      SameTest → (Re[#]^2 + Im[#]^2 ≥ 4 &)]; i], RuntimeAttributes -> {Listable}];
```

```
ArrayPlot[f[Table[x + ⅈ y, {x, -2.1, 0.7, 0.005}, {y, -1.3, 1.3, 0.005}]],
 ColorFunction → "GreenPinkTones"]
```



```
ArrayPlot[f[Table[x + i y, {x, 0, 0.5, 0.001}, {y, 0.3, 1, 0.001}]],
 ColorFunction → "GreenPinkTones"]
```



Compilation: esempi

In molti casi per le funzioni native *Mathematica* utilizza la compilazione automaticamente, ma per le funzioni definite dall'utente i casi in cui riesce ad "identificare ed isolare" porzioni di codice compilabile sono davvero pochi. Pertanto è utile prendere dimestichezza con la funzione **Compile**.

```
Clear[f];
```

```
f[x_] /; NumberQ[x] = Integrate[1/(1 + x^4), x]
```

$$\frac{1}{4\sqrt{2}}\left(-2\,\text{ArcTan}\left[1-\sqrt{2}\,x\right]+2\,\text{ArcTan}\left[1+\sqrt{2}\,x\right]-\text{Log}\left[1-\sqrt{2}\,x+x^2\right]+\text{Log}\left[1+\sqrt{2}\,x+x^2\right]\right)$$

```
Table[f[x], {x, 0, 1, 0.0001}]; // Timing
{0.244663, Null}
```

Proviamo a compilare questa funzione:

```
compiled = Compile[{x}, Evaluate[Integrate[1/(1 + x^4), x]]];

g[x_] /; NumberQ[x] := compiled[x]

Table[g[x], {x, 0, 1, 0.0001}]; // Timing
{0.022452, Null}
```

Compilazione: esempi

Le opzioni della **Compile**. Ci sono tre opzioni che permettono di ottimizare ulteriormente il codice compilato con la **Compile**:

RuntimeAttributes

Parallelization

CompilationTarget

```
n = 10^5;
data = N[Range[n]];
```

```
cf = Compile[{{x, _Real}, {n, _Integer}},
    Module[{y = 1.}, Do[y = (y + x / y) / 2, {n}]; y], RuntimeAttributes → Listable];
```

```
cfp = Compile[{{x, _Real}, {n, _Integer}}, Module[{y = 1.}, Do[y = (y + x / y) / 2, {n}]; y],
    RuntimeAttributes → Listable, Parallelization → True];
```

```
dllfp = Compile[{{x, _Real}, {n, _Integer}}, Module[{y = 1.}, Do[y = (y + x / y) / 2, {n}]; y],
    RuntimeAttributes → Listable, Parallelization → True , CompilationTarget → "C"];
```

```
AbsoluteTiming[Length[r1 = cf[data, n]]]
```

```
{133.710514, 100 000}
```

```
AbsoluteTiming[Length[r2 = cfp[data, n]]]
```

```
{132.804881, 100 000}
```

```
AbsoluteTiming[Length[r3 = dllfp[data, n]]]
```

```
{126.146468, 100 000}
```

◄ | ►

Conclusioni

Sebbene in moltissimi casi *Mathematica* abbia già un livello di ottimizzazione elevato (per cui i suoi algoritmi nativi sono altamente performanti sui moderni hardware), vi sono diverse tecniche - come quelle illustrate in questo seminario - che consentono al codice utente di poter raggiungere livelli di prestazione altamente soddisfacenti. Il tutto senza richiedere, per la maggior parte dei casi, una elevata competenza all'utente stesso.