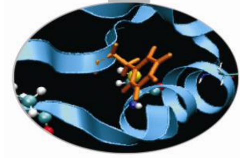
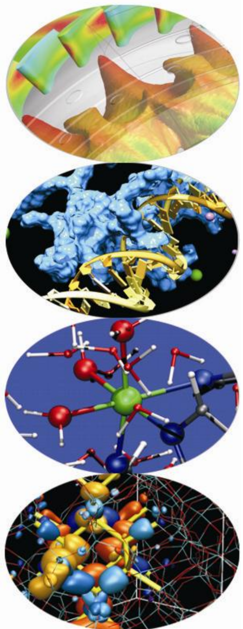
CINECA SCAI
SuperComputing Applications and Innovation

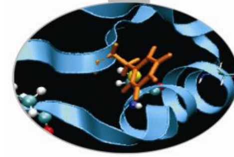


GPU (Graphics Processing Unit) Programming in CUDA

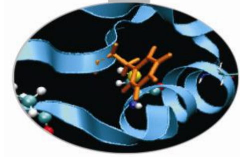


Giorno 3

CINECA



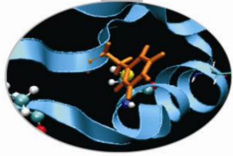
- CUDA Streams
- Ambtiente Multi-GPU



Interazioni GPU-CPU

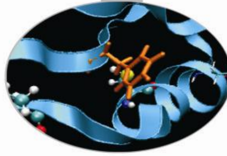
- Interazione GPU-CPU
  - trasferimenti
  - pinned memory
  - uso degli Stream concorrenti
  - multi-GPU

Trasferimento Dati GPU-CPU



- La bandwidth GPU-CPU è di molto inferiore rispetto a quella interna al *device*
  - valori di picco: 8GB GB/s (PCIe x16 Gen 2) o 16GB GB/s (PCIe x16 Gen 3) vs. 144 GB/s (Fermi) o 200GB/s (Kepler)
  - valori reali: 3.6GB/s - 86GB/s (Fermi) o 144 GB/s (Kepler)
- Minimizzare i trasferimenti!
  - eventuali dati intermedi possono/debbono essere allocati, manipolati e deallocati sulla GPU, senza essere copiati sulla memoria della CPU
    - può convenire ricalcolare dei dati sulla GPU piuttosto che trasferirli
- Raggruppare i trasferimenti
  - un singolo trasferimento di grandi dimensioni è preferibile a molti piccoli trasferimenti

Trasferimenti con memoria “locked”

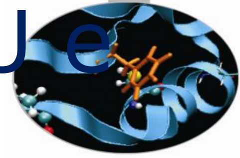


```
• real(kind(0.0d0)), allocatable, dimension(:), pinned :: pinned_mem
```

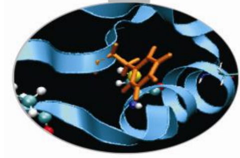
```
• cudaHostAlloc((void**) &pinned_mem, size, cudaHostAllocDefault);
```

- A differenza di `malloc()` in C, o del semplice attributo `allocatable` in FORTRAN, `cudaHostAlloc()` o l’attributo `pinned`, consentono l’allocazione su CPU di memoria non paginabile (*paged-locked* o *pinned*)
  - fino a CUDA 3.0 si usava `cudaMallocHost((void**) &a, size);`
  - impedisce allo scheduler dell’*host* di spostare le pagine *pinned*
  - riduce la memoria fisica disponibile per la paginazione
    - possibile riduzione delle performance dell’*host*
- Migliora le prestazioni della cudaMemcpy
  - circa 3 GB/s su PCI-e x16 Gen1
  - circa 6 GB/s su PCI-e x16 Gen2
  - utilizzare il tool “bandwidthTest” del CUDA SDK
- `cudaFreeHost(a);` libera la memoria allocata
  - ricordarsi di farlo non appena possibile (risorsa preziosa!)

Sovrapposizione di elaborazione GPU e CPU



- Le API CUDA di default sono serializzate:
  - i kernel GPU sono asincroni e ritornano il controllo al codice CPU prima del loro completamento
  - tuttavia i trasferimenti dati tra *host* e *device* (in ambo le direzioni) sono sincrone: bloccano il flusso di codice *host* fino al loro completamento
- Per sovrapporre trasferimento dati a operazioni di calcolo sulla CPU è possibile utilizzare delle API CUDA asincrone oppure ricorrere agli Streams
- le API CUDA asincrone ritornano il controllo di flusso al codice *host* prima del loro completamento
  - permettono di sovrapporre trasferimenti HtoD e DtoH con il calcolo o altre operazioni sulla CPU
- gli Stream CUDA definiscono una sequenza di operazioni CUDA consecutive
  - è possibile definire stream diversi e attribuire a ciascuno un identificativo
  - kernel e trasferimenti definiti in stream diversi possono sovrapporsi
  - se non specificato, tutte le operazioni CUDA appartengono allo stream0



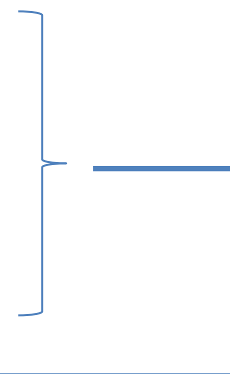
Trasferimento Asincrono

- Il trasferimento dati asincrono restituisce il controllo alla CPU prima del suo completamento

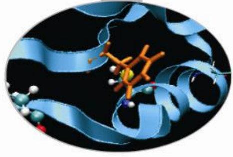
```
cudaMemcpyAsync(dst, src, size, dir, stream);
```

  - se non specificato, lo stream di default è lo 0
  - tutte le operazioni CUDA avvengono sequenzialmente
  - ma ritornano subito il controllo alla CPU
  - l'effetto è quello di sovrapporre l'elaborazione GPU e il trasferimento dati con altre operazioni indipendenti da poter richiamare sulla CPU

```
cudaMemcpyAsync(a_d, a_h, size,
cudaMemcpyHostToDevice);
kernel<<<grid, block>>>(a_d);
cudaMemcpyAsync(a_h, a_d, size,
cudaMemcpyDeviceToHost);
// not using a_h, a_d
cpuFunction();
```



Stream CUDA ed esecuzione concorrente

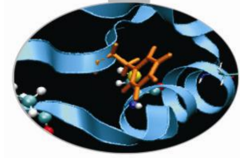


- Richieste:
  - la memoria su host deve essere *pinned*
  - device con compute capability >= 1.1
  - il kernel e il trasferimento devono usare stream diversi
  - Richiede "Concurrent copy and execute" (vedi campo deviceOverlap della struttura cudaDeviceProp)

```
cudaStream_t stream1, stream2;
cudaStreamCreate(&stream1);
cudaStreamCreate(&stream2);

cudaMemcpyAsync(dst, src, size, dir,stream1);
kernel<<<grid, block, 0, stream2>>>(…);
```

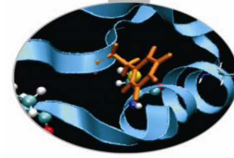
Sovrapposte

Sequenza di esecuzione degli Stream



- Le operazioni CUDA appartenenti a ciascuno stream sono eseguite nel loro ordine di definizione
- Le schede **Fermi** possono eseguire concorrentemente:
  - fino a 16 kernel
  - 2 memcopies in direzioni differenti (DtoH e HtoD)
- Le schede **Kepler** possono eseguire concorrentemente:
  - Fino a 32 kernel
  - 2 memcopies in direzioni differenti (DtoH e HtoD)
- una operazione è eseguita se sono verificate entrambe le seguenti condizioni:
  - le risorse richieste sono disponibili
  - le operazioni precedenti dello stesso stream sono terminate

Sequenza di esecuzione degli Stream



Fermi Model

STREAM 1	STREAM 2	STREAM 3
A | B | C	P | Q | R	X | Y | Z

A-B-C P-Q-R X-Y-Z

Single hardware work queue

Kepler Hyper-Q Model

STREAM 1	STREAM 2	STREAM 3
A | B | C	P | Q | R	X | Y | Z

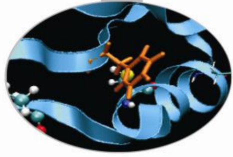
A-B-C

P-Q-R

X-Y-Z

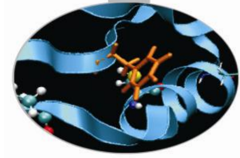
Each stream receives its own work queue

Ambiente Multi-GPU



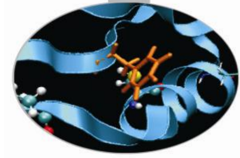
- Le GPU non condividono la memoria globale
  - Una GPU può accedere direttamente la memoria di un'altra GPU se condividono lo stesso bus (vero da cuda toolkit 4.0)
- Comunicazione tra GPU
  - Le comunicazioni fra GPU devono sempre passare attraverso l'host (vero fino al cuda toolkit 3.2)
  - I dati sono scambiati attraverso il PCIe

Ambiente Multi-GPU



- Un thread CPU può controllare un solo “GPU context” per volta
  - L’utilizzo di più GPU richiede più CPU thread
  - Più CPU thread possono definire un contesto con la stessa GPU
- Il driver gestisce il time-sharing ed il partizionamento delle risorse
- Le GPU sono identificate da interi consecutivi a partire da 0

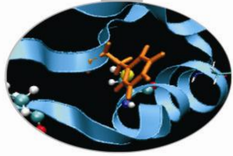
Gestione della GPU



- La CPU può richiedere informazioni e scegliere una GPU con le seguenti primitive

  - **cudaGetDeviceCount** `(int* count)`
  - **cudaSetDevice** `(int device)`
  - **cudaGetDevice** `(int *current_device)`
  - **cudaGetDeviceProperties** `(cudaDeviceProp *prop, int device)`
  - **cudaChooseDevice** `(int* device,cudaDeviceProp* prop)`

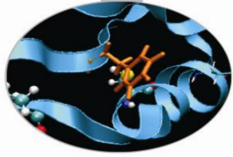
SuperComputing Applications and Innovation



Gestione della GPU: esempio

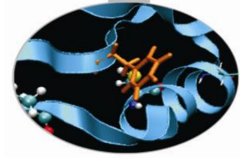
```
cudaGetDevice (&cudadevice);
cudaGetDeviceProperties(&prop,cudadevice);
mname=prop.name;
muva=prop.unifiedAddressing;
mpc=prop.multiProcessorCount;
mtpb=prop.maxThreadsPerBlock;
shmsize=prop.sharedMemPerBlock;
printf("Device %d: number of
      multiprocessors:%d\n"
      "max number of threads per block %d\n"
      "shared memory per block
      %d\n",cudadevice,mpc,mtbp,shmsize)
```

Scelta della GPU



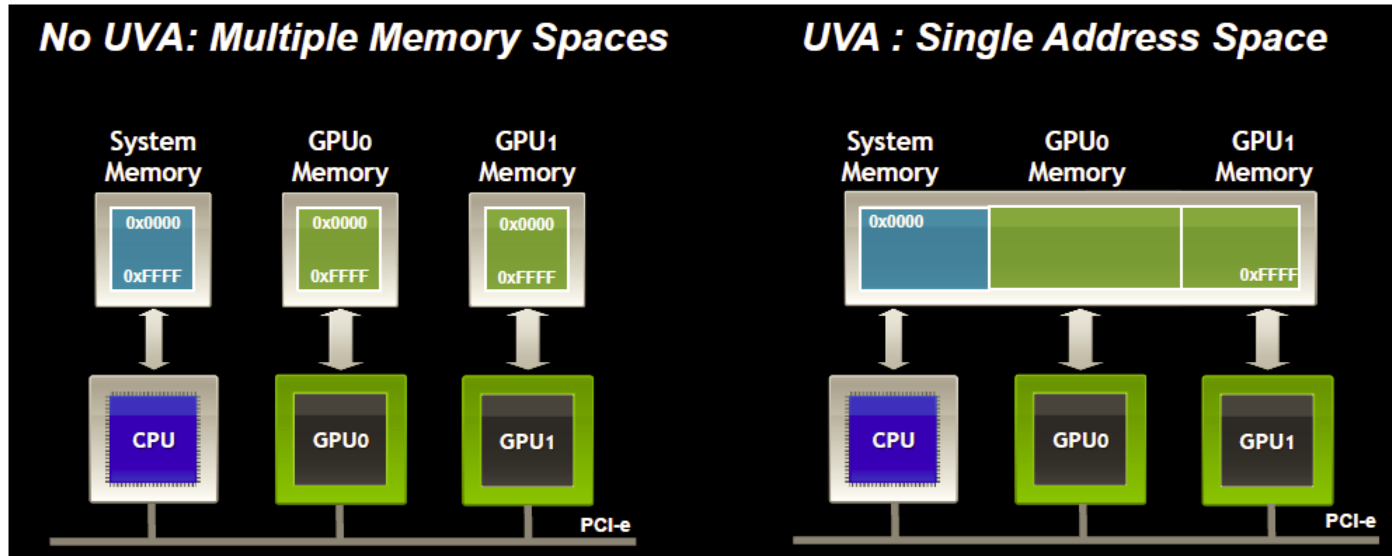
- Scelta esplicita:
  - Si seleziona il device (cioè la GPU) invocando la funzione **`cudaSetDevice(devnum)`**
    - Deve essere chiamata prima della creazione del contesto (i.e., prima di qualsiasi altra primitiva CUDA)
    - È possibile forzare la creazione di un contesto semplicemente con cudaFree(0)
- Scelta implicita:
  - Se non viene chiamata cudaSetDevice, viene scelta la GPU 0

CINECA SCAI
SuperComputing Applications and Innovation



Unified Virtual Addressing

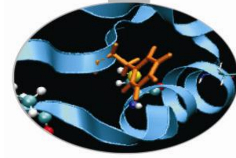
- Indirizzamento virtuale unificato CPU – GPU
  - La memoria lato host deve essere pinned
  - Allocazione lato host con `cudaHostAlloc()`
  - Allocazione lato device con `cudaMalloc()`
  - Copia con `cudaMemcpy()` e parametro **`cudaMemcpyDefault`**
  - I puntatori ritornati da `cudaHostAlloc()` possono essere utilizzati direttamente nei kernel (CUDA zero copy)



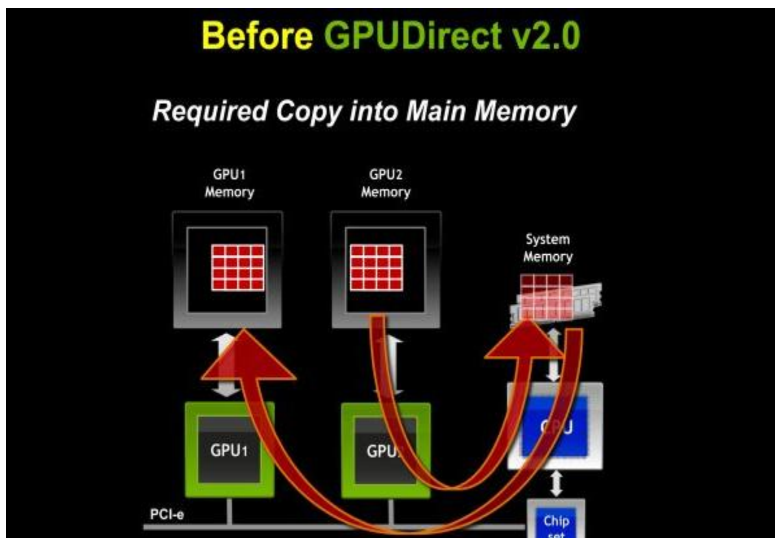
CINECA

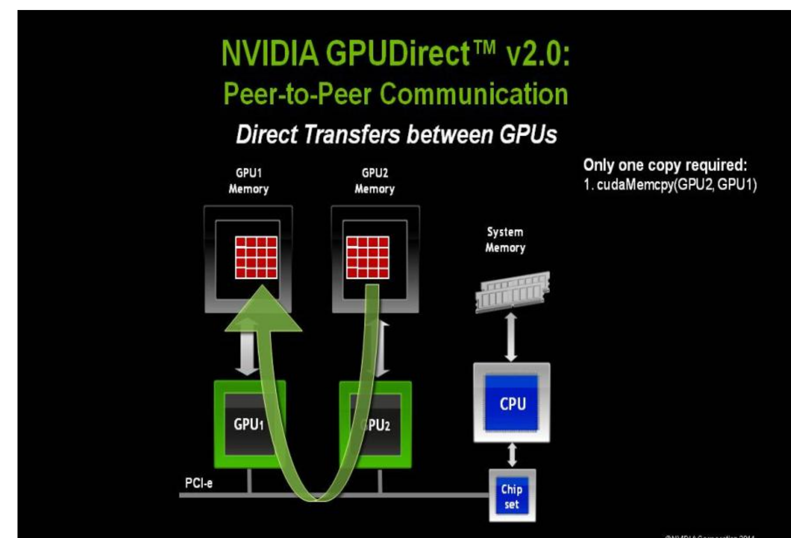
SuperComputing Applications and Innovation

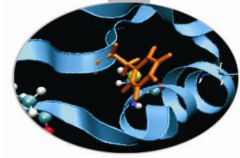
Comunicazione tra GPU



- Comunicazioni Peer-to-Peer
  - Copia diretta GPU-GPU, una sola chiamata `cudaMemcpy()`
  - La modalità Peer-to-Peer deve essere abilitata esplicitamente con `cudaDeviceEnablePeerAccess(int peerdevice, unsigned int flags)`
  - Per verificare che un device disponga di tale capacità si può utilizzare `cudaDeviceCanAccessPeer(int * canAccessPeer, int device, int peerDevice)`





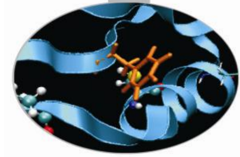


Comunicazione tra GPU

- **Esempio Peer-to-Peer**
  - **Verifica dell'accesso peer-to-peer di due GPU**

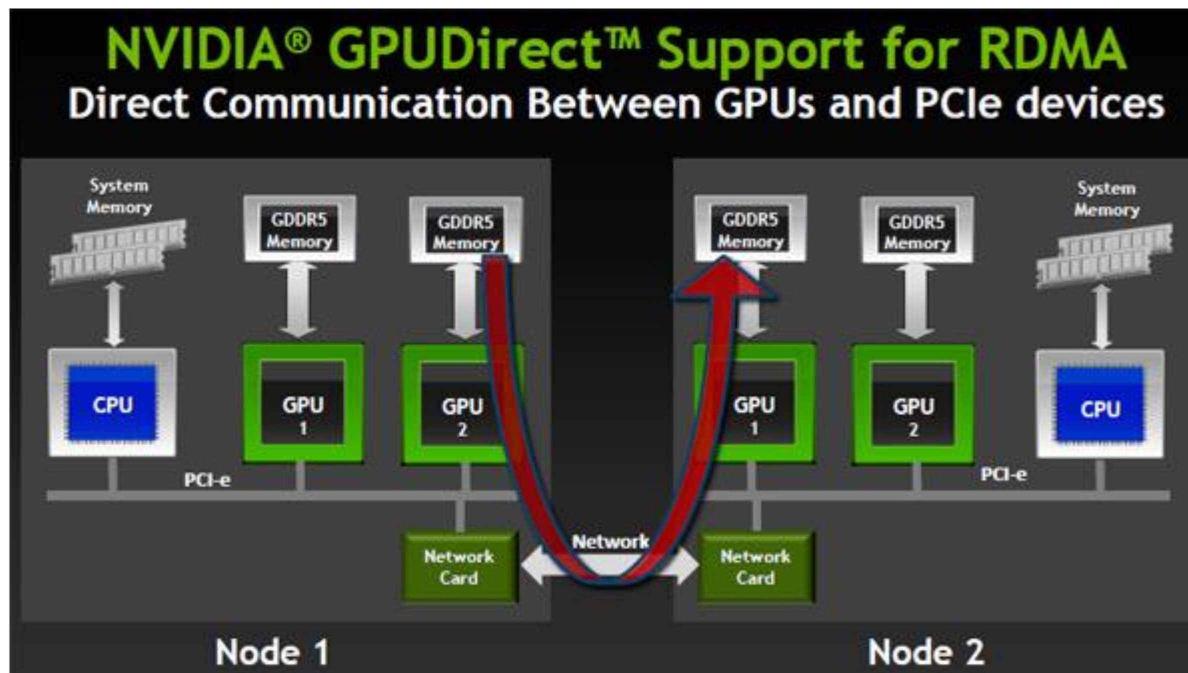
```
cudaDeviceCanAccessPeer(&can_access_peer_0_1, gpuid_tesla[0],
gpuid_tesla[1]);
cudaDeviceCanAccessPeer(&can_access_peer_1_0, gpuid_tesla[1],
gpuid_tesla[0]);
```

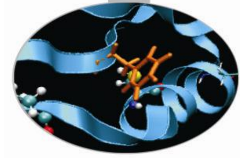
  - **Abilitazione accesso peer-to-peer**
  - `cudaDeviceEnablePeerAccess(gpuid_tesla[0],0);`
  - `cudaDeviceEnablePeerAccess(gpuid_tesla[1],0);`
  - **Allocazione U array g0 su GPU0**
  - `cudaSetDevice(gpuid_tesla[0]);`
  - `cudaMalloc(&g0, buf_size);`
  - **Allocazione array g1 su GPU1**
  - `cudaSetDevice(gpuid_tesla[1]);`
  - `cudaMalloc(&g1, buf_size);`
  - **Copia array g0 su GPU0 in array g1 su GPU1 e viceversa**
  - `cudaMemcpy(g1, g0, buf_size, cudaMemcpyDefault);`
  - `cudaMemcpy(g0, g1, buf_size, cudaMemcpyDefault);`



Comunicazione tra GPU

- Comunicazioni GPUDirect RDMA
  - A partire da CUDA 5.0 La tecnologia GPUDirect permette la comunicazione diretta tra GPU e altri dispositivi PCI-E presenti nel sistema; supporta l'accesso diretto alla memoria tra la GPU e le schede di rete.
  - La comunicazione tra le GPU avviene tramite chiamate MPI dove al buffer vengono passati gli indirizzi di memoria della GPU.





Comunicazione tra GPU

- **Codice senza CUDA RDMA ed integrazione con MPI**
  - Mittente

```
cudaMemcpy(s_buf, s_device, size, cudaMemcpyDeviceToHost);
MPI_Send(s_buf, size, MPI_CHAR, 1, 1, MPI_COMM_WORLD);
```

  - Ricevente

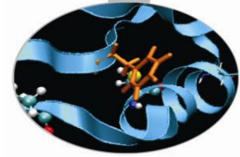
```
MPI_Recv(r_buf, size, MPI_CHAR, 0, 1, MPI_COMM_WORLD, &req);
cudaMemcpy(r_device, r_buf, size, cudaMemcpyHostToDevice);
```

- **Codice con CUDA RDMA ed integrazione con MPI**
  - Mittente

```
MPI_Send(s_device, size, ...);
```

  - Ricevente

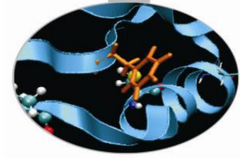
```
MPI_Recv(r_device, size, ...);
```



Esercitazione

Esercitazione

- Stream
- Peer-to-peer + UVA
- MPI +CUDA

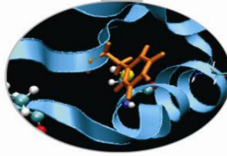


Esercizi

Stream

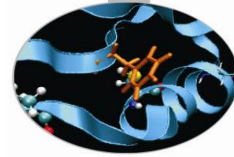
- Completare il programma simpleStream.cu di modo che :
  - venga inizializzata la struttura dati "deviceProp" tramite la funzione "cudaGetDeviceProperties"
  - si utilizzi "deviceProp" per verificare di avere una compute capability maggiore di 1.1 e si esca dal programma in caso contrario. Se la compute capability è maggiore di 1.1 si ponga la variabile "niterations" uguale a 5
  - copiare il vettore host "a" di "nbtes" dati nel vettore device d_a usando cudaMemcpyAsync sullo stream 0
  - lanciare il kernel "init_array" sullo stream zero
  - lanciare il kernel "init_array" e la copia da devicea host di "d_a" in "a" senza utilizzare gli streams
  - chiamare il kernel su tutti gli streams dividendo il vettore "d_a" di dimensione "n" in parti uguali per ogni stream
  - copiare in modo asincrono la corrispettiva parte per ogni stream del vettore "d_a" in "a"

Librerie



- Librerie:
  - cuBLAS, cuFFT
  - cenni su cuDPP e Thrust
- Applicazioni CUDA-enabled
  - NAMD, AMBER, GROMACS
  - *more to come!*
- Esercitazione:
  - cuBLAS

CINECA SCAI
SuperComputing Applications and Innovation



Come programmare su GPU?

Applicazione

Librerie | **Direttive** | **Linguaggi di programmazione** | **High level Language**

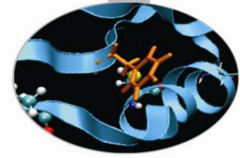
Easiest Approach

Maximum performance

No need of programming skill

CINECA

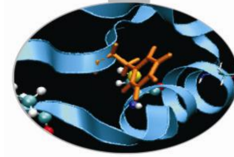
OpenACC direttive



Direttive del compilatore per specificare regioni parallele:

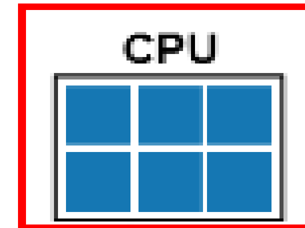
- Codice portabile
- Le regioni parallele sono eseguite su GPU
- Non necessita di trasferimenti espliciti
- Non necessita di inizializzazioni esplicite
- Consente di ottenere accelerazione hardware in modo semplice e a costi ridotti

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA. The Portland Group



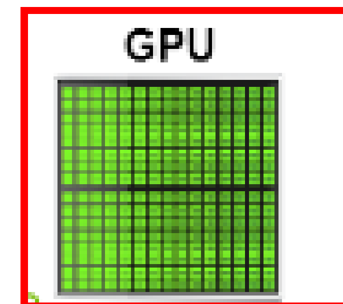
OpenACC direttive

```
#include <stdio.h>
#define N 1000000

 int main(void) {
  double pi = 0.0f;
  long i;
  #pragma acc parallel loop
  for (i=0; i<N; i++) {
     double t= (double)((i+0.5)/N);
     pi +=4.0/(1.0+t*t);
  }
  printf("pi=%16.15f\n",pi/N);
  return 0;
  }
```

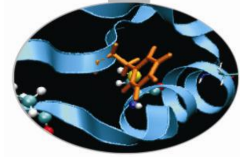


Original Code

Compiler Hint



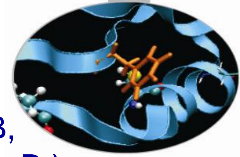
Moltiplicazione Matriciale CUDA /OpenACC



```
void computeMM(float *c, float *a, float *b,int n){

double t_start = omp_get_wtime();
#pragma acc data region copyin(a[0:n*n-1],b[0:n*n-1]) copyout(c[0:n*n-1]){
// PERFORM MULTIPLICATION
// loop over output rows
#pragma acc region   {
#pragma acc for independent
for ( int row=0; row<n; row++ ) {
//loop over output columns
            #pragma acc for independent
            for ( int col=0; col<n; col++ ) {
                        // initialize output result to zero
                        double val = 0;
                        // loop over inner dimension
                                    for ( int k=0; k<n; k++ ) {
                                                val += a[row*n+k] * b[k*n+col]; }
            c[row*n+col] = val;    }  }  }}
// compute elapsed time
double et = omp_get_wtime() - t_start;
```

Direttive

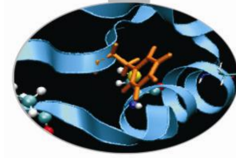
Moltiplicazione Matriciale CUDA /OpenACC



```
__global__ void matrixMul( double* C, double* A,
double* B, int wA, int wB)
 {
 int bx = blockIdx.x;
 int by = blockIdx.y;
int tx = threadIdx.x;
 int ty = threadIdx.y;
 int aBegin = wA * BLOCK_SIZE * by;
 int aEnd = aBegin + wA - 1;
 int aStep = BLOCK_SIZE;
 int bBegin = BLOCK_SIZE * bx;
 int bStep = BLOCK_SIZE * wB;
 float Csub = 0;
 for (int a = aBegin, b = bBegin;a <= aEnd; a += aStep,
b += bStep) {
 __shared__ double As[BLOCK_SIZE][BLOCK_SIZE];
 __shared__ double Bs[BLOCK_SIZE][BLOCK_SIZE];
 AS(ty, tx) = A[a + wA * ty + tx];
 BS(ty, tx) = B[b + wB * ty + tx];
 __syncthreads();
 for (int k = 0; k < BLOCK_SIZE; ++k)
 Csub += AS(ty, k) * BS(k, tx);
 __syncthreads();
 }
 int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
 C[c + wB * ty + tx] = Csub;
}
```

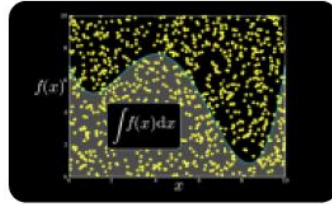
```
void domatmul( double* C, double* A, double* B,
unsigned int hA, unsigned int wA , unsigned int wB )
 {
 unsigned int size_A = WA * HA;
 unsigned int mem_size_A = sizeof(float) * size_A;
 unsigned int size_B = WB * HB;
 unsigned int mem_size_B = sizeof(float) * size_B;
 unsigned int size_C = WC * HC;
 unsigned int mem_size_C = sizeof(float) * size_C;
 float *d_A, *d_B, *d_C;
 cudaMalloc((void**) &d_A, mem_size_A);
 cudaMalloc((void**) &d_B, mem_size_B);
 cudaMalloc((void**) &d_C, mem_size_C);
 cudaMemcpy(d_A, h_A, mem_size_A,
cudaMemcpyHostToDevice);
 cudaMemcpy(d_B, h_B, mem_size_B,
cudaMemcpyHostToDevice);
 dim3 threads(BLOCK_SIZE, BLOCK_SIZE);
 dim3 grid(WC / threads.x, HC / threads.y);

 matrixMul<<< grid, threads >>>(d_C, d_A, d_B, WA,
WB);

 cudaMemcpy(h_C, d_C, mem_size_C,
cudaMemcpyDeviceToHost);
 cudaFree(d_A);
 cudaFree(d_B);
 cudaFree(d_C);}
```

CUDA C

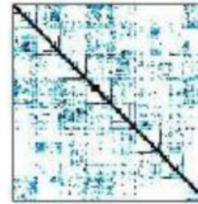


Librerie CUDA

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

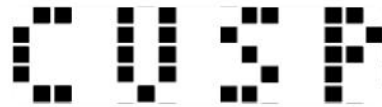
CULA | tools

GPU Accelerated Linear Algebra

Matrix Algebra on GPU and Multicore

NVIDIA cuFFT

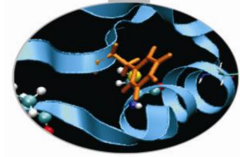
IMSL Library



Sparse Linear Algebra

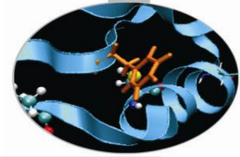
Building-block Algorithms for CUDA

C++ STL Features for CUDA

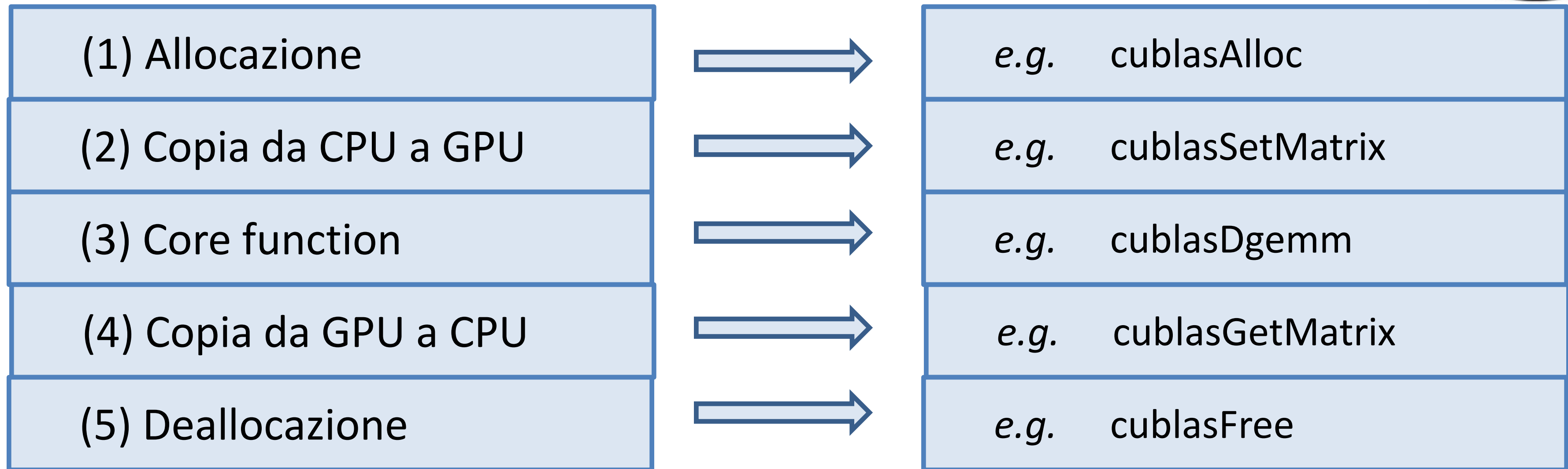


CUBLAS: CUDA Basic Linear Algebra Subroutine

- Funzioni Blas:
  - Livello 1: vettore-vettore O(N)
  - Livello 2: matrice-vettore O(N^2)
  - Livello 3: matrice-matrice O(N^3)
- Implementazione autocontenuta a livello di API, nessuna interazione con driver CUDA
- Da CUBLAS 3.0 implementate tutte le BLAS, precisione singola (S) doppia (D), complessa singola (C) e complessa doppia (Z)
- Interfaccia alla CUBLAS in cublas.h
- Applicazioni devono linkare cublas.so (Linux), cublas.dll (Windows) or cublas.dylib (Mac OS X)
- Possibilità di emulazione usando cublasemu.so, cublasemu.dll e cublasemu.dylib



CUBLAS: schema di utilizzo

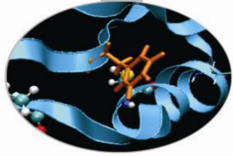


- Funzioni helper:
  - Allocazione della memoria e trasferimento dati
  - Vettori e matrici così allocati possono essere usati altrove (cublasAlloc è un wrapper di cudaMalloc e cublasSetMatrix un wrapper di cudaMemcpy!). Disponibili versioni di copia asincrona.

- Funzioni core : la convenzione per i nomi è:

  cublas + BLAS name ⟹ e.g., cublasSGEMV

CUBLAS: trattamento dell'errore



- Le funzioni helper ritornano direttamente l'errore

```
status = cublasAlloc(n2, sizeof(d_A[0]), (void**)&d_A);
```

- Le funzioni core non ritornano errore: si utilizza allora cublasGetError() per accedere l'ultimo errore

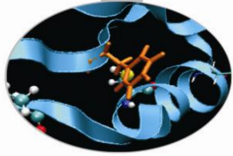
```
status = cublasGetError()
```

- Il check dell'errore può avvenire come di consueto

```
if (status != CUBLAS_STATUS_SUCCESS) {
              fprintf (stderr, "!!!! device memory
  allocation error (A)\n");
              return EXIT_FAILURE;
```

SuperComputing Applications and Innovation

CUBLAS: uso con linguaggio C

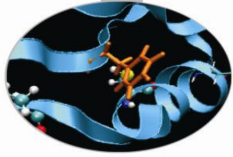


- L'ordinamento delle matrici è column-major e 1-index based: i programmatori C devono lavorare con puntatori o array 1D ed utilizzare eventualmente macro per riordinare

```
 #define IDX2F(i,j,ld) ((((j)-1)*(ld))+((i)-1))
for (j = 1; j <= N; j++) {
    for (i = 1; i <= M; i++) {
        a[IDX2F(i,j,M)] = (i-1) * M + j;
    }
}
```

- Per non rinunciare alla semantica di array multidimensionale si può considerare di chiamare le funzioni per le matrici trasposte

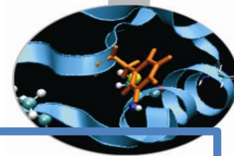
CUBLAS:esempio in Cuda C SSCAL



```
void modify (float *m, int ldm, int n, int p, int q, float
alpha, float beta)
{
        cublasSscal (n-q+1, alpha, &m[IDX2F(p,q,ldm)], ldm);
        cublasSscal (ldm-p+1, beta, &m[IDX2F(p,q,ldm)], 1);
}
```

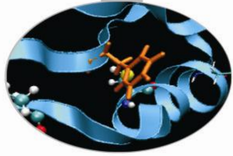
```
cublasInit();
stat = cublasAlloc (M*N, sizeof(*a), (void**)&devPtrA);
stat = cublasSetMatrix (M, N, sizeof(*a), a, M, devPtrA, M);
modify (devPtrA, M, N, 2, 3, 16.0f, 12.0f);
stat = cublasGetMatrix (M, N, sizeof(*a), devPtrA, M, a, M);
cublasShutdown();
```

CUBLAS: esempio in Cuda C SGEMV (1)



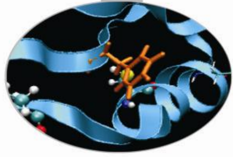
```
#include "cublas.h"
/* Allocate device memory for the matrices */
  status = cublasAlloc(n2, sizeof(d_A[0]), (void**)&d_A);
  if (status != CUBLAS_STATUS_SUCCESS) {
      fprintf (stderr, "!!!! device memory allocation error (A)\n");
      return EXIT_FAILURE;
  }
  status = cublasAlloc(N, sizeof(d_B[0]), (void**)&d_B);
  if (status != CUBLAS_STATUS_SUCCESS)   { fprintf (...); return ... }
  status = cublasAlloc(N, sizeof(d_C[0]), (void**)&d_C);
  if (status != CUBLAS_STATUS_SUCCESS)   { fprintf (...); return ... }
/* Initialize the device matrices with the host matrices */
  status = cublasSetVector(n2, sizeof(h_A[0]), h_A, 1, d_A, 1);
  if (status != CUBLAS_STATUS_SUCCESS)  { fprintf (...); return ... }
  status = cublasSetVector(N, sizeof(h_B[0]), h_B, 1, d_B, 1);
   .....
  status = cublasSetVector(N, sizeof(h_C[0]), h_C, 1, d_C, 1);
```

CUBLAS: esempio in Cuda C SGEMV (2)



```
/*   Performs operation using cublas */
    cublasSgemv('n', N, N, alpha, d_A, N, d_B, 1, beta, d_C, 1);
    status = cublasGetError();
    if (status != CUBLAS_STATUS_SUCCESS) {
      fprintf (stderr, "!!!! kernel execution error.\n");
      return EXIT_FAILURE;
    }
    status = cublasGetVector(N, sizeof(h_C[0]), d_C, 1, h_C, 1);
    status = cublasFree(d_A);
    status = cublasFree(d_B);
    status = cublasFree(d_C);
```

CUBLAS: uso con linguaggio Fortran



Per chiamare le CUBLAS dal Fortran ci sono 3 possibilità:

1. Wrapper (stile fortran 77)
2. Interfacce e modulo iso_c_binding (stile fortran 95/2003)
3. PGI CudaFortran

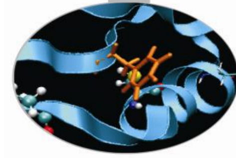
Wrapper: Nvidia fornisce i wrapper *fortran.c* e *fortran_thunking.c*. Occorre compilare prima questi, e.g.

```
gcc -O3 -I$CUDADIR/include/ -c fortran.c
```

Poi linkare con il proprio programma, e.g. fortran.o.

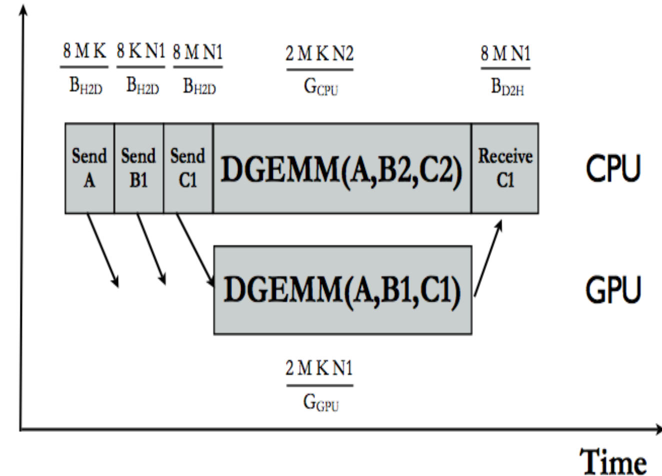
```
gfortran -O3 code.f90 fortran.o -L$CUDADIR/lib64  -lcublas
```

Attenzione: i wrapper contengono solo istruzioni cublas. Se si deve fare altro (anche solo una sincronizzazione) occorre scrivere a mano i wrapper o utilizzare altri metodi!



CUBLAS + BLAS

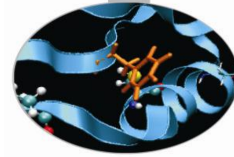
- CPU e GPU possono lavorare insieme
- Per il caso di CPU + 1 GPU l'implementazione si avvale del fatto che i kernel restituiscono subito il controllo alla CPU: chiamando la blas dopo la cublas si ha che il calcolo viene effettuato in parallelo.




```
call time(loc_T0)
ierr = cublasDGEMM
    (one,two,N,N_gpu,N,alpha,devPtrA,N,devPtrB,N,beta,devPtrC,N)
call
    DGEMM('n','n',N,N_cpu,N,alpha,A,N,B(1:N,N_gpu+1:N),N,beta,C(1:N,N_gpu
    +1:N),N)
 ierr = cudaThreadSynchronize()
 call time(loc_T1)
```

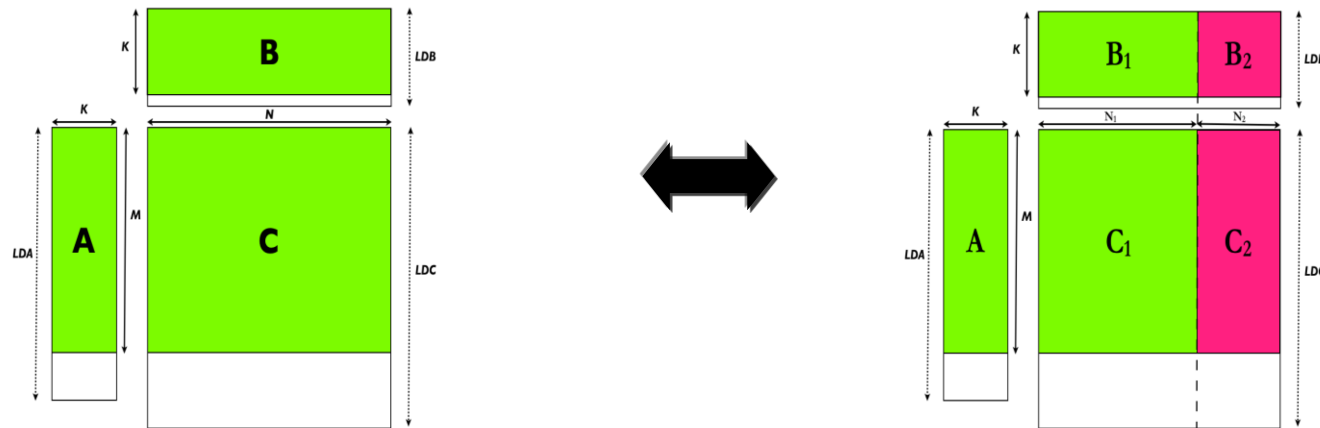
- Per il caso multi-CPU + multi-GPU occorre qualche paradigma di parallelizzazione: e.g. MPI

SCAI
SuperComputing Applications and Innovation



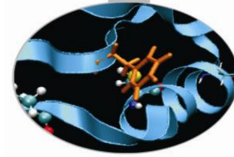
CUBLAS+BLAS

- La divisione dei compiti dovrebbe essere il più possibile bilanciata, cioè proporzionale alle capacità di calcolo CPU e GPU
- Se η = T_GPU/T_CPU allora 1/η = N_GPU/N_CPU



- η dipende dalle taglie della matrice, quindi è un problema non lineare
- in prima approssimazione tenere conto di taglie più comode per suddividerla,
e.g. 4096=3072+1024 può essere meglio di una η fissata a priori

CINECA

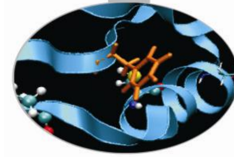


CUBLAS DGEMM performance

Performance cluster JAZZ: prodotto matrice-matrice:
size 4096x4096; doppia precisione

blas + cublas MPI
blas + cublas
cublas 3.2 (magmablas 0.3)
cublas 3.1
shared memory hand-made GPU
naive hand-made GPU (cache a 48Kb)
PGI accelerator
mkl, 12 thread
unrolling+padding, ifort -fast
naive 12 thread, ifort -fast
mkl
unrolling+padding, ifort -fast
naive, ifort -fast

0 50 100 150 200 250 300 350 400 450



CUBLAS DGEMM Performance

Performance DGEMM Xeon Quad-Core con Tesla C1060.
Dipendenza dalla taglia della matrice.

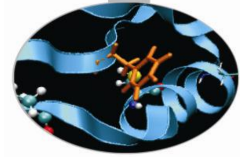
GFLOPs

120
100
80
60
40
20
0

128 320 512 704 896 1088 1280 1472 1664 1856 2048 2240 2432 2624 2816 3008 3200 3392 3584 3776 3968 4160 4352 4544 4736 4928 5120 5312 5504 5696 5888 6080

Size

Xeon Quad-core 2.8 GHz, MKL 10.3
Tesla C1060 GPU (1.296 GHz)
CPU + GPU

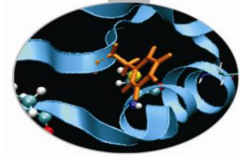


CUFFT

- CUFFT è la libreria CUDA per le FFT parallele

- Le API CUFFT sono modellizzate su quelle FFTW
  - Basate sul concetto di “piano”, che specifica completamente la configurazione ottimale per eseguire la FFT di un certo tipo e dimensione

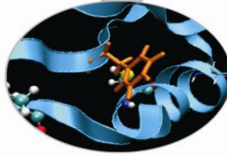
- Una volta che è stato creato un “piano”, la libreria mantiene le informazioni necessarie ad eseguire più volte il piano senza ricalcolare la configurazione
  - Modello molto efficace perché i diversi tipi di FFT hanno bisogno di diverse configurazioni di thread e risorse GPU



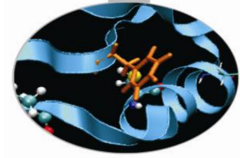
Caratteristiche della CUFFT

- Trasformate 1D, 2D e 3D di dati reali e complessi
- Per le trasformate 2D e 3D, CUFFT esegue le trasformate in ordine row-major (come in C)
  - Attenzione se viene invocata da FORTRAN o MATLAB!
- Trasformata in-place e out-of-place.
- Direzione
  - CUFFT_FORWARD (-1) and CUFFT_INVERSE (1)
    - secondo il segno del termine esponenziale complesso
- Le parti reale ed immaginaria dei vettori di input e output sono alternate
  - Viene definito un tipo specifico cufftComplex a questo scopo

CuFFT esempio: trasformata 2D complex-complex



```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);
…
/* Crea un piano FFT 2D */
 cufftPlan2d(&plan, NX,NY, CUFFT_C2C);
 /* Usa il piano CUFFT per trasformare il segnale "out of place" */
 cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);
/* Trasformata inversa del segnale "in place" */
 cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);
/* Puntatori ad input ed output diversi implicano trasformate "out of place"
*/
 /* Elimina il piano CUFFT  */
   cufftDestroy(plan);
   cudaFree(idata), cudaFree(odata);
```

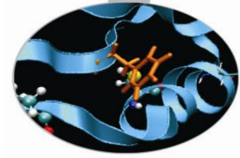


CUDPP

- CUDPP: CUDA Data Parallel Primitives library
- Libreria di primitive parallele per:
  - prefix-sum (“scan”)
  - sort
  - reduction
  - generazione di numeri Random
  - grafi, alberi
- Caratteristiche
  - Scritta in C
  - Supporto per Windows, Linux e OS X
  - Elevate performance

SCAI
SuperComputing Applications and Innovation

CUSPARSE

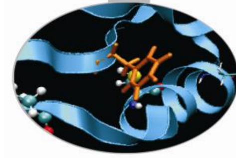


- cuSparse contengono un set di subroutine utilizzate per la gestione di matrici sparse:
  - Livello 1: operazioni tra un vettore sparso e un vettore denso
  - Livello 2: operazioni tra una matrice sparse e un vettore denso
  - Livello 3: operazioni tra una matrice sparse e un insieme di vettori densi
  - Routines di conversione

E' pensata per essere interfacciata con applicazioni scritte in C/C++. Esiste tuttavia un wrapper per codice Fortran.

CINECA

CUSPARSE



Coordinate (COO)

Row Index 1 2 2 3 4 4 4

Col Index 1 1 2 3 1 3 4

Values 1.0 2.0 3.0 4.0 5.0 6.0 7.0

1.0			
2.0	3.0		
		4.0	
5.0		6.0	7.0

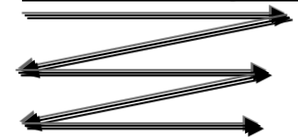
Compressed Row Format (CSR)

Row Index 1 2 4 5 8

Col Index 1 1 2 3 1 3 4

Values 1.0 2.0 3.0 4.0 5.0 6.0 7.0

Row major



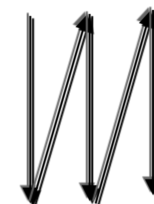
Compressed Row Format (CSC)

Row Index 1 2 4 2 3 4 4

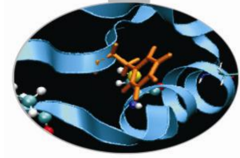
Col Index 1 2 4 5 7 8

Values 1.0 2.0 3.0 4.0 5.0 6.0 7.0

Column major



CUSPARSE



Sparse vector storage

Index 1 4 6

Values 1.0 2.0 3.0

1 2 3 4 5 6

[1.0 0.0 0.0 2.0 0.0 3.0]

Convenzione per i nomi di funzione:

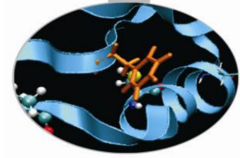
```
cusparse<Type>[<sparse data
   format>]<operation>[<sparse data format>]
```

- Esempio:

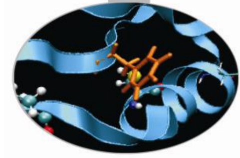
  Moltiplicazione matrice (csr storage) – vettore in singola precisione
  `cusparseScsrmv`

  Moltiplicazione matrice (csr storage) – matrice in doppia precisione
  `cusparseDcsrmm`

CUSPARSE



- Level 1 (x vettore sparso e y vettore denso y)
  - axpyi (vector add) — y = y + alpha x
  - doti (dot product) — alpha = y^T* x
  - dotci (conjugate dot product) — alpha = y^H* x
  - gthr (gather) — x = y
  - sctr (scatter) — y=x
  - roti (Givens rotation) — [x,y]=[x,y] [c-s
s c]



CUSPARSE

- Livello 2 e 3:

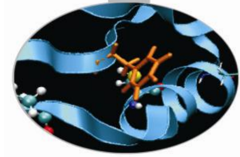
csrmv (matrix- vector multiplication)

$$\begin{bmatrix} x1 \\ x2 \\ x3 \\ x4 \end{bmatrix} = \alpha \begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \beta \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

csrtrsv (triangular solve)

$$\begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \alpha \begin{bmatrix} 10.0 \\ 20.0 \\ 30.0 \\ 40.0 \end{bmatrix}$$

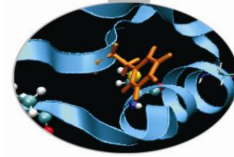
CINECA SCAI SuperComputing Applications and Innovation



CUSPARSE

- Conversione tra tipi:
  - nnz
  - dense2csr
  - dense2csc
  - csr2dense
  - csc2dense
  - csr2coo
  - coo2csr
  - csr2csc

CINECA



CULA

- CULA sono un set di librerie CUDA per l'algebra matriciale
- Hanno un interfaccia semplice ed avanzata che permette l'interfacciamento con più linguaggi di programmazione: C,C++,Fortran,Python, Matlab
- Si dividono in :

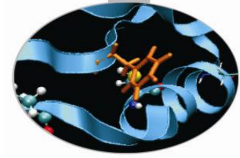
CULA Dense

Forniscono una versione accelerata delle librerie LAPACK/BLAS per sistemi densi.

- Solver diretti di sistemi lineari
- Decomposizione a valori singolari
- Fattorizzazioni

CULA Sparse

- Solver iterativi per sistemi sparsi
- Precondizionatori
- Supporto per diversi formati



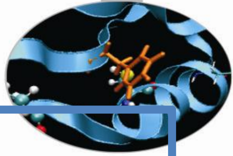
Thrust

- Thrust è una libreria CUDA di algoritmi paralleli con una interfaccia simile alla STL di C++
  - nasconde chiamate dirette a kernel CUDA
  - implementa Containers generici:
    - thrust::host_vector<T>
    - thrust::device_vector<T>
    - compatibili con i container STL (vector, list, map, ecc)
  - e Algoritmi:
    - thrust::sort(), thrust::reduce(), thrust::inclusive_scan(), ecc

Sito Ufficiale: **http://code.google.com/p/thrust/**

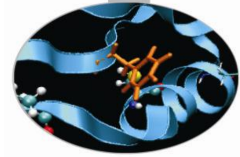
(Jared Hoberock and Nathan Bell di NVIDA Research)

Thrust by examples



```
// generate 32M random numbers on the host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(), h_vec.end(), rand);


// transfer data to the device
thrust::device_vector<int> d_vec = h_vec;

// sort data on the device (846M keys per second on GeForce GTX 480)
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(), d_vec.end(), h_vec.begin());
```

```
// generate random data on the host
 thrust::host_vector<int> h_vec(100);
thrust::generate(h_vec.begin(), h_vec.end(), rand);

// transfer to device and compute sum
thrust::device_vector<int> d_vec = h_vec;
int sum = thrust::reduce(d_vec.begin(), d_vec.end);

// obtain raw pointer to device vector's memory
 int * ptr = thrust::raw_pointer_cast(&d_vec[0]);
// use ptr in a CUDA C kernel
 my_kernel<<<N/256, 256>>>(N, ptr);
```



Applicazioni CUDA enabled

APPLICATIONS

Molecular Dynamics & Quantum Chemistry

- ACE MD
- AMBER
- BigDFT (ABINIT) (news)
- GROMACS
- HOOMD
- LAMMPS
- NAMD
- TeraChem (Quantum Chemistry)
- VMD

Bio Informatics

- CUDA-BLASTP
- CUDA-EC
- CUDA-MEME
- CUDASW++ (Smith-Waterman)
- DNADist
- GPU Blast
- GPU-HMMER
- HEX Protein Docking
- Jacket (MATLAB Plugin)
- MUMmerGPU
- MUMmerGPU++

Complex molecular simulations that had been only possible using supercomputing resources can now be run on an individual workstation, optimizing the scientific workflow and accelerating the pace of research.

http://www.nvidia.com/object/tesla_computing_solutions.html

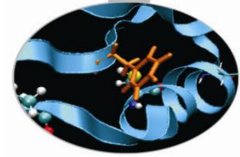
http://www.nvidia.com/object/tesla_testimonials.html

http://www.nvidia.com/object/cuda_home_new.html ← CUDA Zone

La lista delle applicazioni HPC "CUDA-enabled" è in continua espansione.

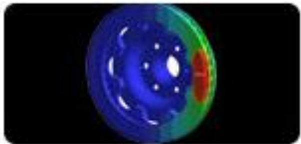
In media si ottengono incrementi di prestazione (speed-up) che variano da 5 a 20

Applicazioni CUDA enabled



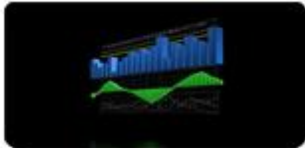
A range of applications now take advantage of the tremendous computing capability of NVIDIA CUDA-enabled GPUs.

See how NVIDIA Tesla solutions are transforming computational research in fields such as:


Computational Structural Mechanics

Bio-Informatics and Life Sciences

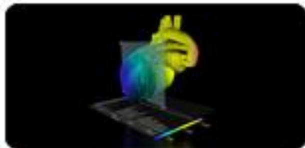
Computational Electromagnetics and Electrodynamics

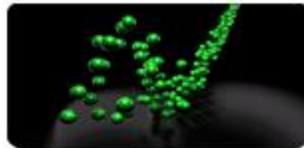

Computational Finance

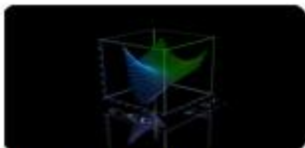
Computational Fluid Dynamics

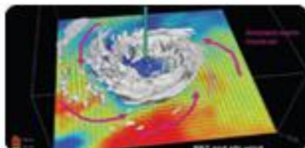
Data Mining, Analytics, and Databases

Imaging and Computer Vision


Medical Imaging


Molecular Dynamics


Numerical Analytics


Weather, Atmospheric, Ocean Modeling, and Space Sciences

Chimica Computazionale

1. NAMD
2. Amber
3. DL-POLY
4. CP2K

Dinamica dei fluidi

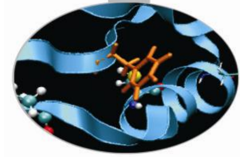
1. OpenFOAM
2. ACUSIM

Analisi Numerica

1. MatLab
2. Matlab+GPU

Bioinformatica

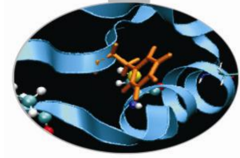
1. CUDA-BLASTP
2. GPU-HMMER
3. CUDASW++



Esercitazione

Esercitazione

- piApproximation, con cuRand
- Serie
- MatrixMul con cuBLAS



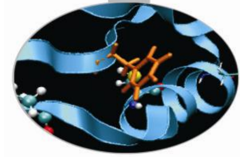
Esercizi

- **piApproximation**
  - Modificare l'implementazione dell'algoritmo di approssimazione del PI generando i numeri casuali su GPU, tramite la libreria CURAND, tramite i seguenti step:
    - Creazione del generatore `curandCreateGenerator()`
    - Definizione del seed del generatore con `curandSetPseudoRandomGeneratorSeed()`
    - Generazione numeri random con `curandGenerate()`
    - Deallocazione risorse tramite `curandDestroyGenerator()`

- **Serie**
  - Implementare un kernel CUDA per la serie di Fourier

$$(\ ) \qquad \frac{(\qquad)}{}$$

Utilizzare le funzioni ,



Esercizi

- Tramite l'utilizzo della libreria CUBLAS implementare il prodotto matriciale.

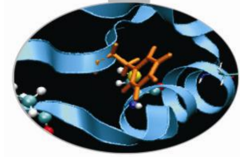
- Utilizzare la funzione cublasSGemm:

void cublasSgemm (char transa, char transb, int m, int n,int k, float alpha, const float *A, int lda,const float *B, int ldb, float beta,float *C, int ldc)

C = alpha * op(A) * op(B) + beta * C

op X ( ) X = op X ( ) XT

- Input
  - transa specifies op(A). If transa == 'N' or 'n', .If transa == 'T', 't', 'C', or 'c', .
  - transb specifies op(B). If transb == 'N' or 'n', .If transb == 'T', 't', 'C', or 'c', .
  - m number of rows of matrix op(A) and rows of matrix C; m must be at least zero.

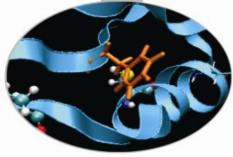


Esercizio

- n number of columns of matrix op(B) and number of columns of C;
- n must be at least zero.
- k number of columns of matrix op(A) and number of rows of op(B);k must be at least zero.
- alpha single precision scalar multiplier applied to . A single precision array of dimensions (lda, k) if transa == 'N' or 'n', and of dimensions (lda, m) otherwise. If transa == 'N' or'n', lda must be at least max(1, m); otherwise, lda must be at least max(1, k).
- lda leading dimension of two-dimensional array used to store matrix A. B single precision array of dimensions (ldb, n) if transb == 'N' or 'n', and of dimensions (ldb, k) otherwise. If transb == 'N' or 'n', ldb must be at least max(1, k); otherwise, ldb must be at least max(1, n).
- ldb leading dimension of two-dimensional array used to store matrix B.
- beta single precision scalar multiplier applied to C. If zero, C does not have to be a valid input. C single precision array of dimensions (ldc, n);
- ldc must be at least max (1, m).ldc leading dimension of two-dimensional array used to store matrix C.

SuperComputing Applications and Innovation

Bibliografia



- CUDA C Programming Guide
- PGI CUDA fortran, http://www.pgroup.com/doc/pgicudaforug.pdf
- CUDA C Best Practices Guide
- Tuning CUDA Applications for Fermi
- Kirk and Hwu, Programming Massively Parallel Processors
- CUDA by example, http://developer.nvidia.com/object/cuda-by-example.html
- P. Micikevicius, Fundamental and Analysis-Driven Optimization, GPU Technology Conference 2010 (GTC 2010)
- V. Volkov, Better performance at lower occupancy, GPU Technology Conference 2010 (GTC 2010)
- J. Dongarra et al. “An Improved MAGMA GEMM for Fermi GPUs”