Introduction to Scientific Programming using GPGPU and CUDA

Day 2

Luca Ferraro
l.ferraro@cineca.it

Sergio Orlandini
s.orlandini@cineca.it

- Tools del CUDA-Toolkit
  - Profiler
  - CUDA-GDB
  - CUDA-memcheck
  - Parallel NSight
- Librerie CUDA-Enabled
  - CUBLAS
  - CUFFT
  - CUSPARSE
  - CURAND
  - MAGMA, THRUST, CUDDP, …
- Esercizi

Profiling tools: built-in

- Il CUDA toolkit mette a disposizione degli utili strumenti di profiling

```
export CUDA_PROFILE=1
export CUDA_PROFILE_CONFIG=$HOME/.config
```

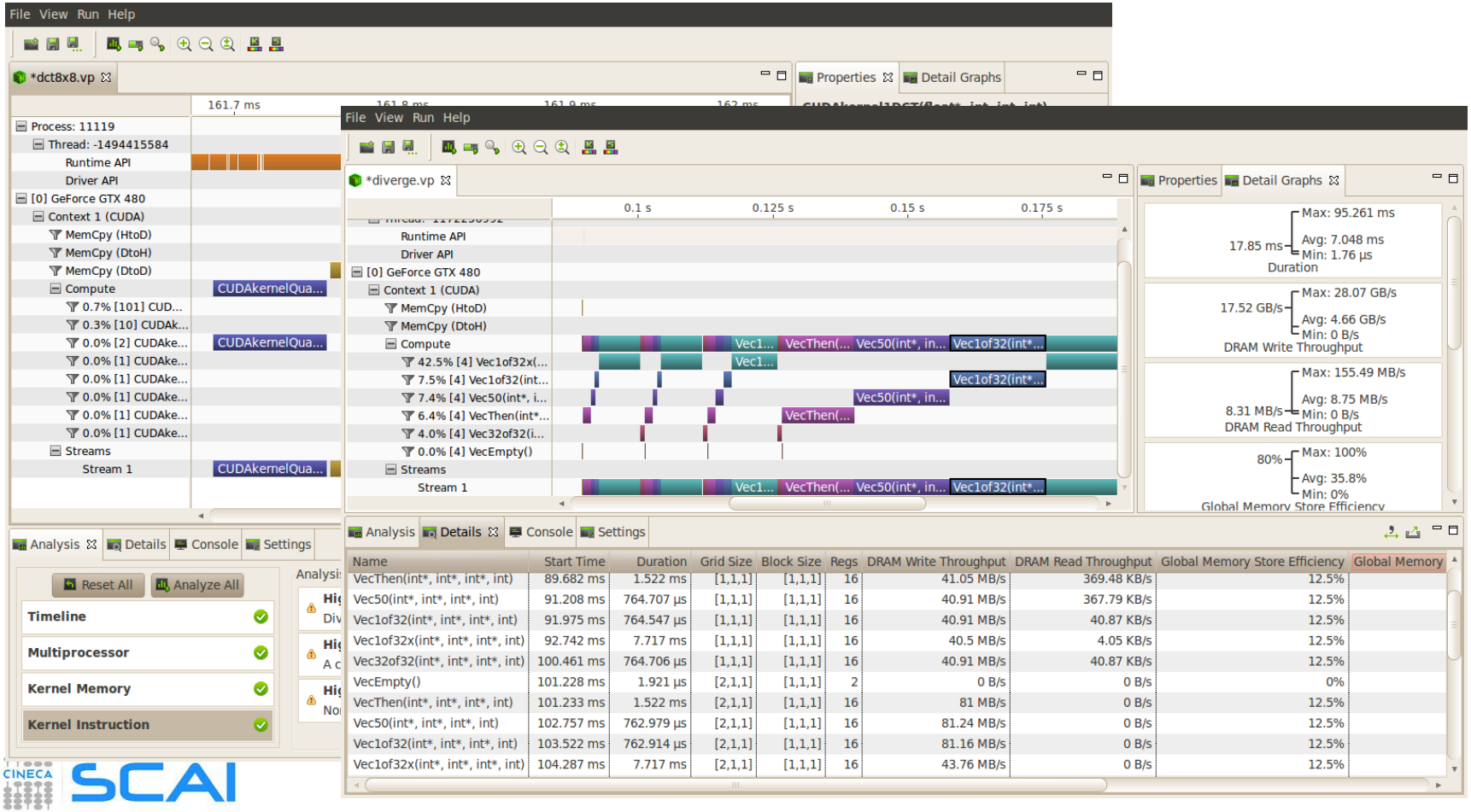
```
// Contents of config
gld_coherent
gld_incoherent
gst_coherent
gst_incoherent
```

```
gld_incoherent: Number of non-coalesced global memory loads
gld_coherent: Number of coalesced global memory loads
gst_incoherent: Number of non-coalesced global memory stores
gst_coherent: Number of coalesced global memory stores
local_load: Number of local memory loads
local_store: Number of local memory stores
branch: Number of branch events taken by threads
divergent_branch: Number of divergent branches within a warp
instructions: instruction count
warp_serialize: Number of threads in a warp that serialize
based on address conflicts to shared or constant memory
cta_launched: executed thread blocks
```

```
method,gputime,cputime,occupancy,gld_incoherent,gld_coherent,gst_incoherent,gst_coherent
method=[ memcopy ] gputime=[ 438.432 ]
method=[ _Z17reverseArrayBlockPiS_ ] gputime=[ 267.520 ] cputime=[ 297.000 ] occupancy=[ 1.000 ]
gld_incoherent=[ 0 ] gld_coherent=[ 1952 ] gst_incoherent=[ 62464 ] gst_coherent=[ 0 ]
method=[ memcopy ] gputime=[ 349.344 ]
```

Profiling: Visual Profiler

- Traccia su HOST le chiamate al driver, trasferimenti, kernel e mostra gli overlap tra stream di esecuzione differenti
- Analisi automatica delle prestazioni (hardware counters)



Debugging: CUDA-GDB

- Debugger tradizionale gdb con estensioni CUDA

```
(cuda-gdb) info cuda threads
BlockIdx ThreadIdx To BlockIdx ThreadIdx Count Virtual PC Filename Line
Kernel 0* (0,0,0) (0,0,0) (0,0,0) (255,0,0) 256 0x0000000000866400
bitreverse.cu 9
(cuda-gdb) thread
[Current thread is 1 (process 16738)]
(cuda-gdb) thread 1
[Switching to thread 1 (process 16738)]
#0 0x000019d5 in main () at bitreverse.cu:34
34 bitreverse<<<1, N, N*sizeof(int)>>>(d);
(cuda-gdb) backtrace
#0 0x000019d5 in main () at bitreverse.cu:34
(cuda-gdb) info cuda kernels
Kernel Dev Grid SMs Mask GridDim BlockDim Name Args
0 0 1 0x00000001 (1,1,1) (256,1,1) bitreverse data=0x110000
```

Debugging: CUDA-MEMCHECK

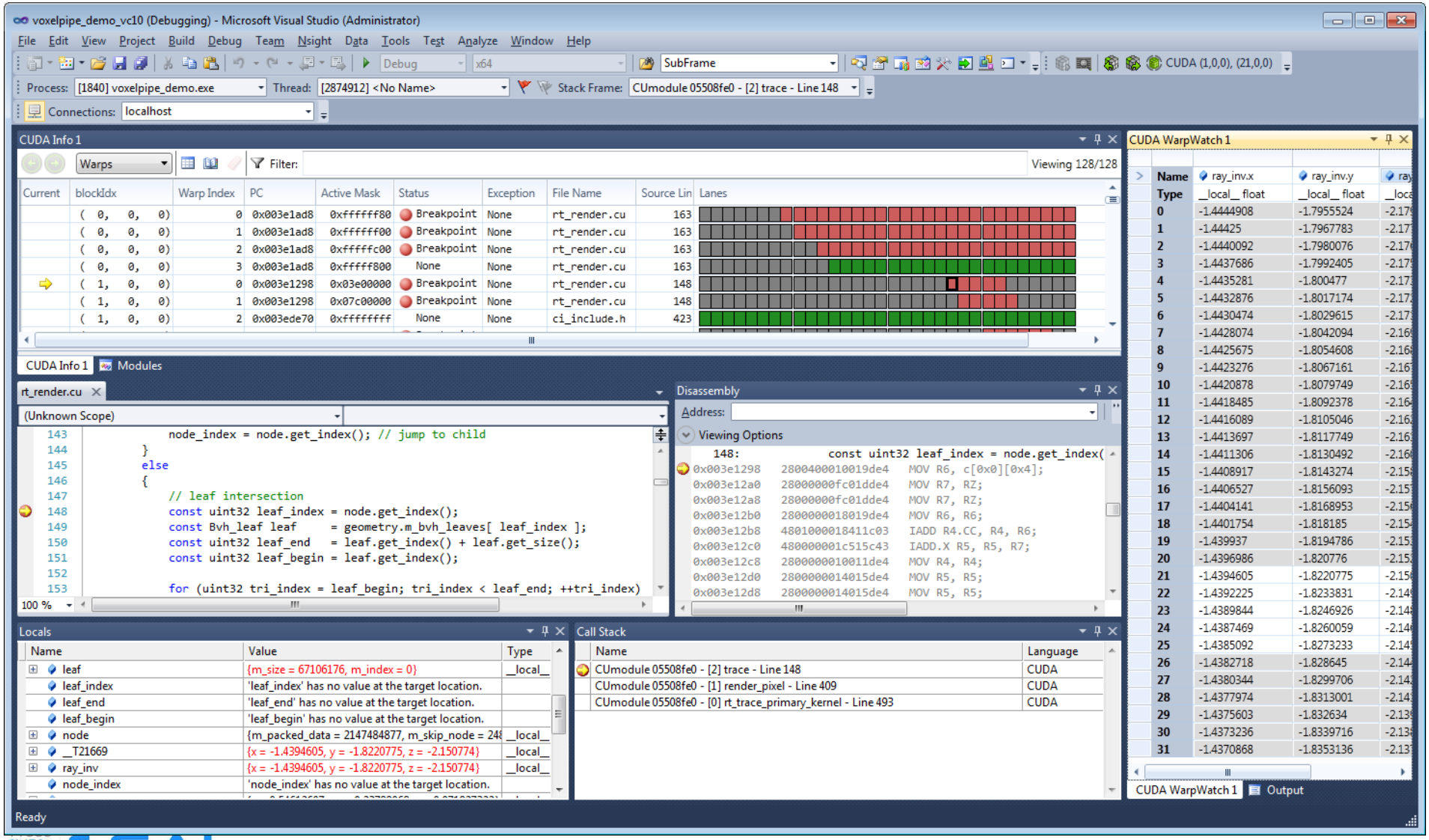
- Tool capace di controllare buffer overflows, misaligned global memory accesses e memory leaks
- Richiamabile anche da CUDA-GDB

```
$ cuda-memcheck --continue ./memcheck_demo
========= CUDA-MEMCHECK
Mallocing memory
Running unaligned_kernel
Ran unaligned_kernel: no error
Sync: no error
Running out_of_bounds_kernel
Ran out_of_bounds_kernel: no error
Sync: no error
========= Invalid __global__ write of size 4
========= at 0x00000038 in memcheck_demo.cu:5:unaligned_kernel
========= by thread (0,0,0) in block (0,0,0)
========= Address 0x200200001 is misaligned
=========
========= Invalid __global__ write of size 4
========= at 0x00000030 in memcheck_demo.cu:10:out_of_bounds_kernel
========= by thread (0,0,0) in block (0,0,0)
========= Address 0x87654320 is out of bounds
=========
=========
========= ERROR SUMMARY: 2 errors
```

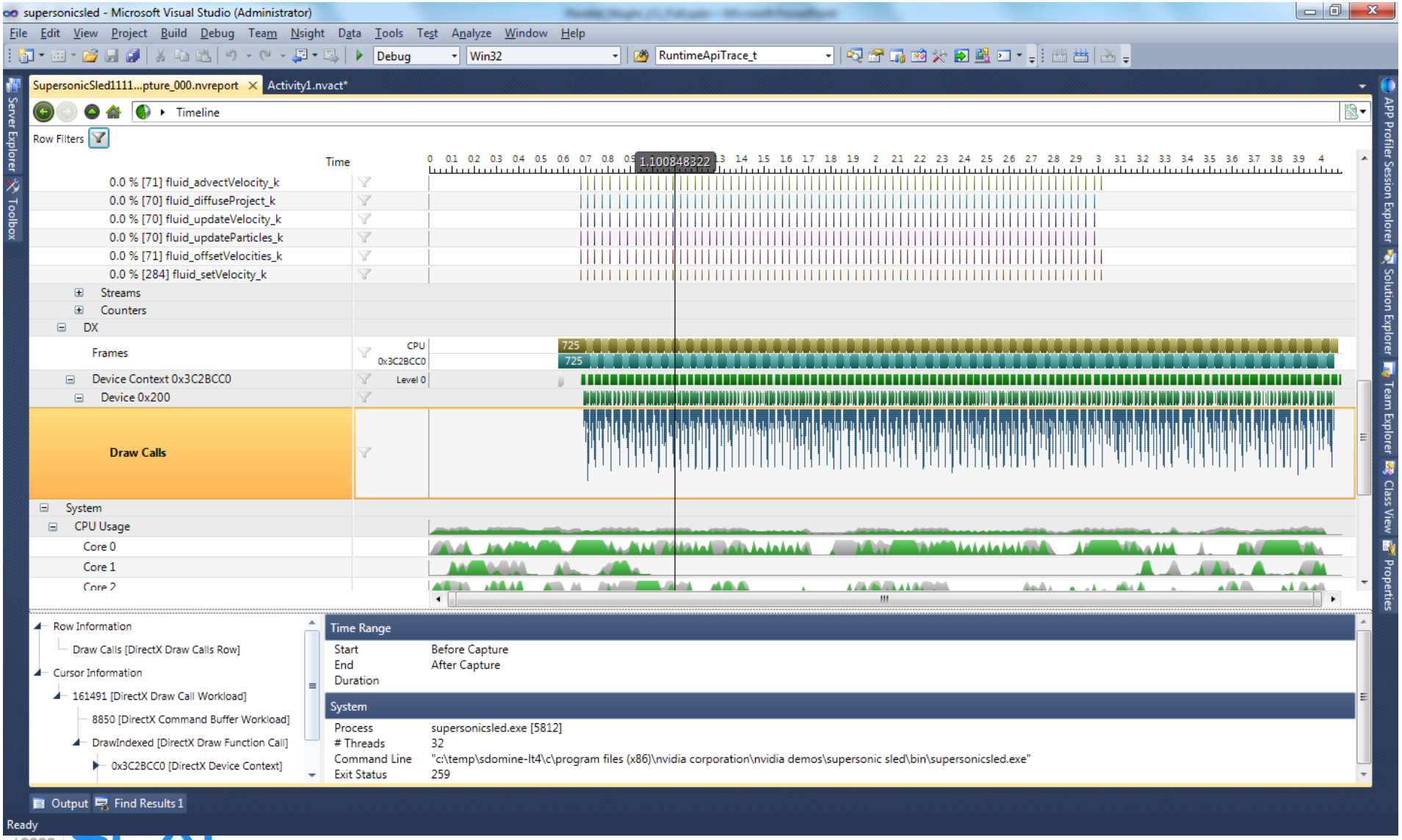
Parallel NSight

- Plug-in disponibile per diversi IDEs (Eclipse, VisualStudio)
- Aggrega diverse funzionalità:
  - Debugger (fully integrated)
  - Visual Profiler
  - Memory correctness checker
- Come plug-in, estende le funzionalità degli IDE per CUDA
- Tool in veloce evoluzione che consente:
  - Debug remoto e profiling
  - PTX assembly view,
  - warp inspector
  - expression lamination

Parallel NSight



Parallel NSight



Librerie CUDA

- CUDA include librerie matematiche di uso comune:
  - CUBLAS: Basic Linear Algerba Subprograms
  - CUFFT : Fast Fourier Transform
  - CUSPARSE: algebra lineare per matrici sparse
  - CURAND: numeri pseudorandom e quasirandom
- Altre librerie matematiche disponibili per CUDA :
  - MAGMA (Matrix Algebra on GPU and Multicore Architectures)
    http://icl.cs.utk.edu/magma/
  - THRUST (CUDA library for parallel algorythms in C++)
    http://code.google.com/p/thrust/
  - CUDPP (Data Parallel Primitives): parallel prefix-sum, sort, reduction
    http://code.google.com/p/cudpp/
  - CULA: Lapack, free solo in singola precisione
    http://www.culatools.com/contact/cuda-training/
  - NPP (Nvidia Performance Primitives) : image e video processing, funzioni statistiche (media, deviazione standard ...)
    http://developer.nvidia.com/object/npp_home.html

CUDA Math Library

- simply by adding "`#include math.h`" in your source code
- Complete support for all C99 standard float and double math functions
- IEEE-754 accurate for float, double, and all rounding modes
- Extended Trigonometry and Exponential Functions
  - cospi, sincos, sinpi, exp10
- Additional Inverse Error Functions
  - erfinv, erfcinv
- Optimized Reciprocal Functions
  - rsqrt, rcbrt
- Floating Point Data Attributes
  - signbit, isfinite, isinf, isnan
- Bessel Functions
  - j0,j1,jn,y0,y1,yn

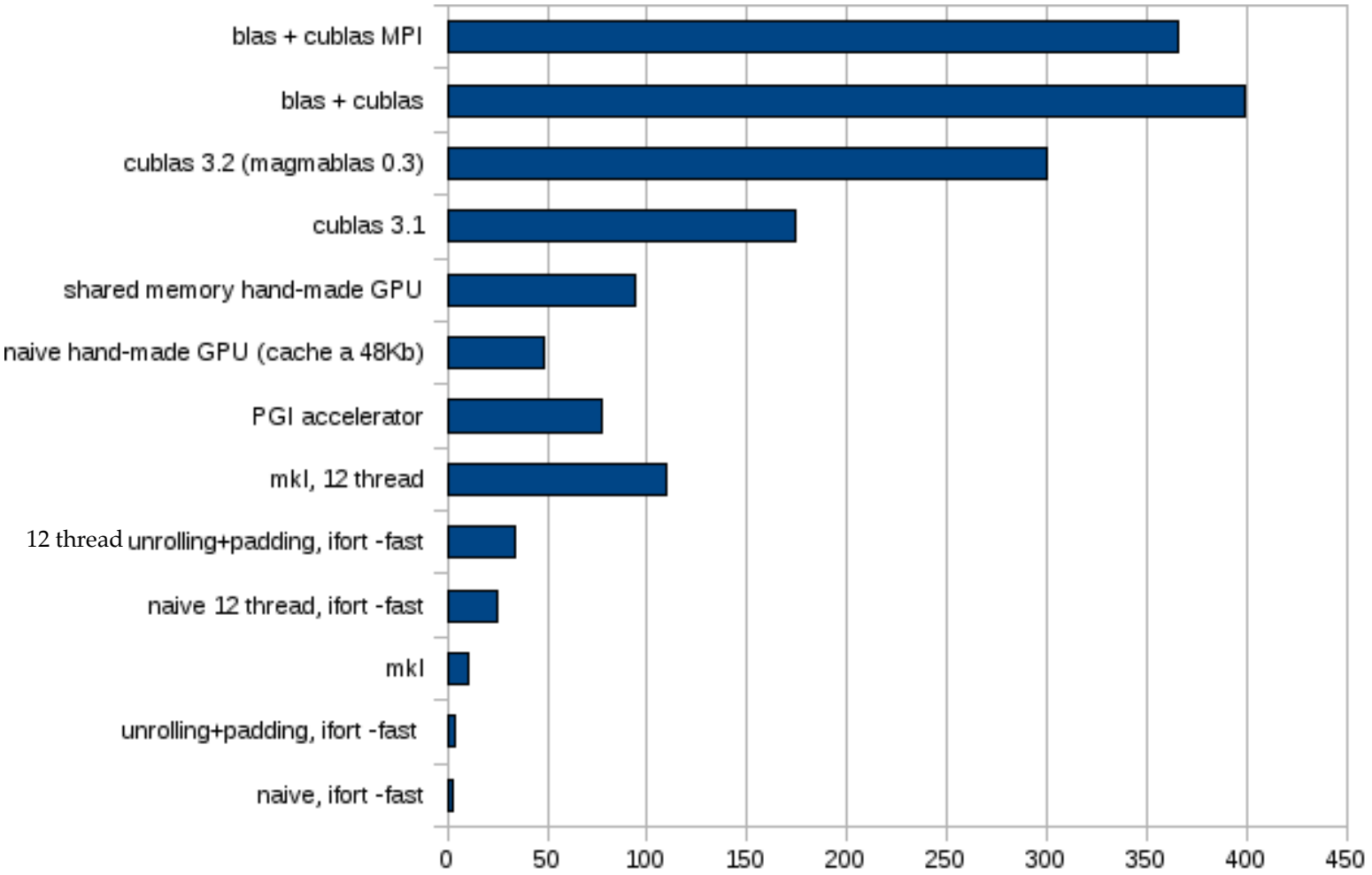
CUBLAS: CUDA Basic Linear Algebra Subprograms

- La libreria BLAS è lo standard accettato per quanto riguarda accuratezza e interfaccia delle librerie che implementano operazioni base di algebra lineare
  - BLAS Level 1: $\mathbf{y} \leftarrow \alpha \boldsymbol{x} + \boldsymbol{y}$ **O(N)**
  - BLAS Level 2: $\mathbf{y} \leftarrow \alpha A \boldsymbol{x} + \beta \boldsymbol{y}$ **O(N^2)**
  - BLAS Level 3: $C \leftarrow \alpha A B \boldsymbol{x} + \beta C$ **O(N^3)**
- Sono disponibili implementazioni vendor delle BLAS ottimizzate per diverse architetture
- La libreria CUBLAS è un'implementazionedelle BLAS basata sul CUDA runtime
- É sviluppata e supportata direttamente da NVIDIA ed è inclusa nel CUDA Toolkit
- Le funzioni CUBLAS supportano il calcolo su una singola GPU (le routine CUBLAS non sono in grado di auto-parallelizzare il calcolo su più GPU)
- Il layout dei dati segue la convenzione FORTRAN (column-major)
- É responsabilità dell'utente la gestione della memoria sul *device*
- La libreria CUBLAS mette a disposizione delle helper function per semplificare la gestione del CUBLAS runtime e il trasferimento dei dati host/device

CUBLAS: DGEMM performance

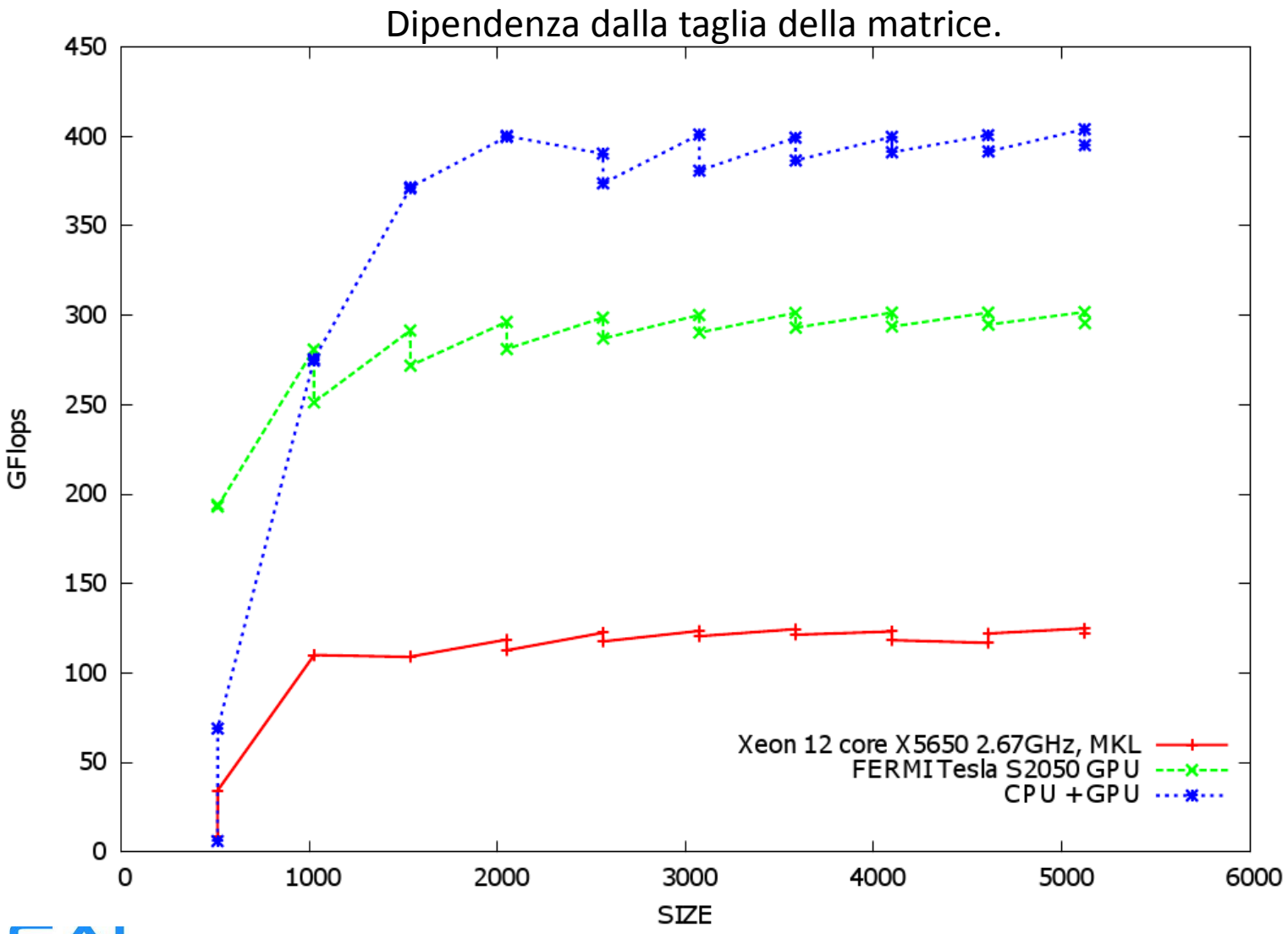
- Performance cluster con FERMI: prodotto matrice-matrice:

size 4096x4096; doppia precisione



CUBLAS: DGEMM performance

- Performance DGEMM cluster FERMI: prodotto matrice-matrice



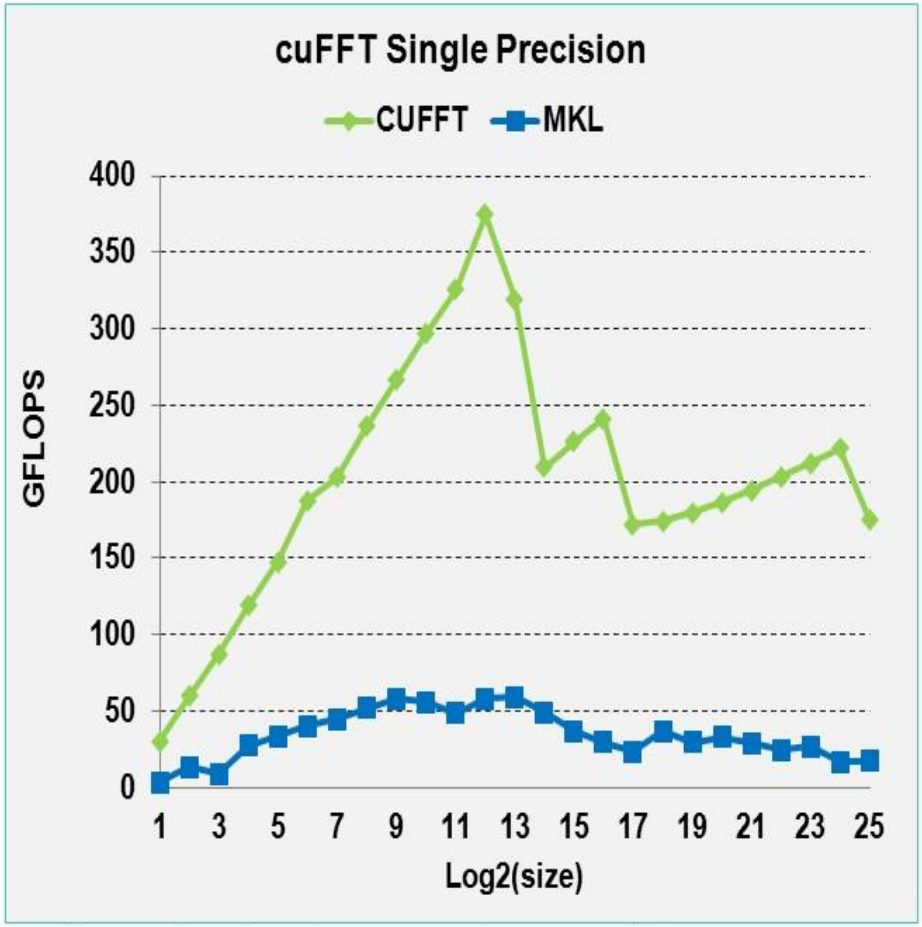
CUFFT

- CUFFT è la libreria per la Fast Fourier Transform fornita da NVIDIA
- FFT basata sugli algoritmi Cooley-Turkey e Bluestein
- L' API fornita è simile all' interfaccia avanzata di FFTW
  - analogamente a FFTW, CUFFT usa il concetto di *workplan*
  - Una volta che è stato creato un *workplan*, 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
  - attenzione, CUFFT segue la convenzione C per il layout in memoria delle matrici (row-major)
- Altre caratteristiche chiave:
  - trasformate 1D, 2D, 3D per tipi di dato reale e complesso
  - supporto a dati in singola e doppia precisione
  - supporto all' esecuzione asincrona e in stream
  - thread-safe (**CUDA 4.1**)

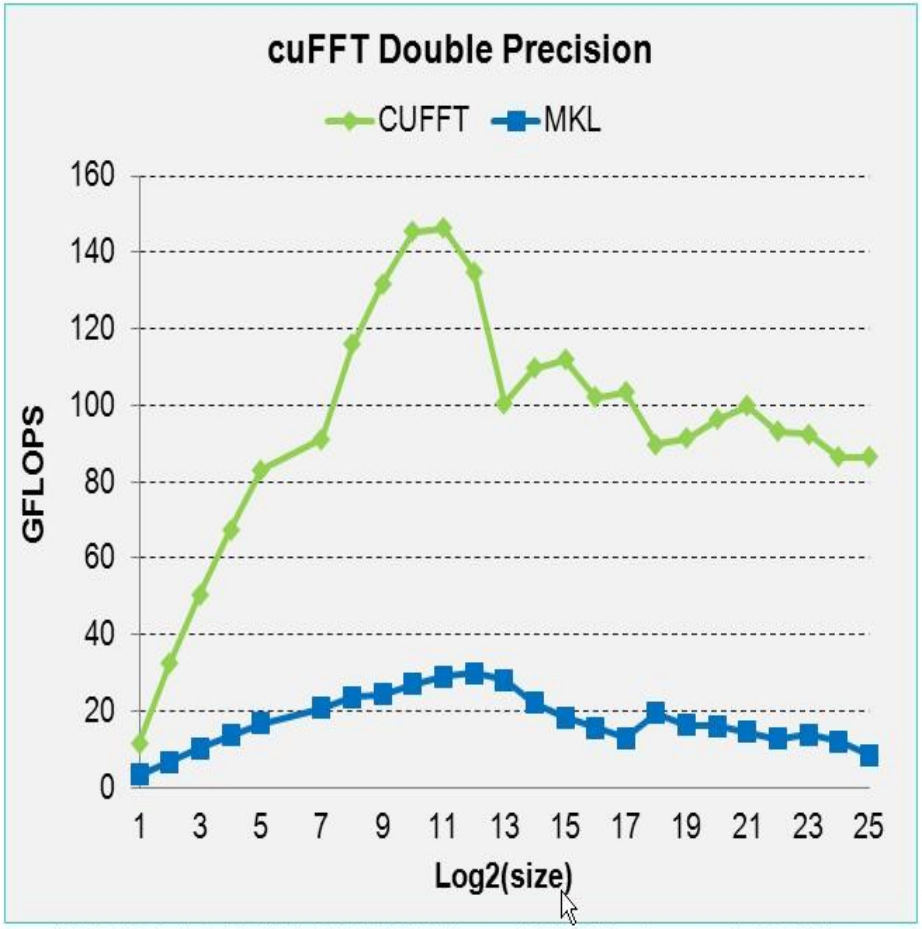
CUFFT esempio: trasformata 2D complex-complex

```
#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);
```

CUFFT: prestazioni FFT1D

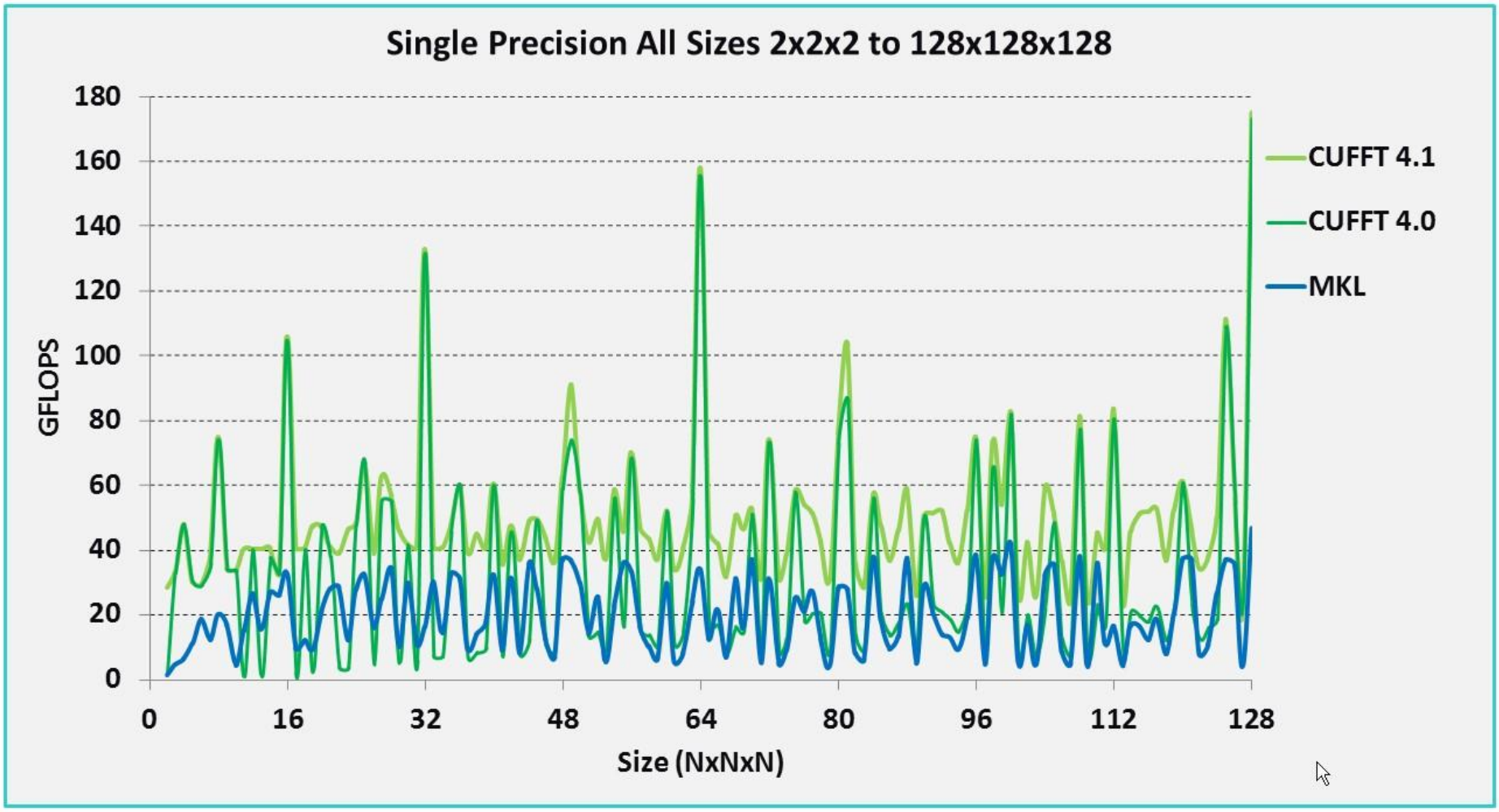


- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz



- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
- Performance may vary based on OS version and motherboard configuration

CUFFT: prestazioni FFT3D

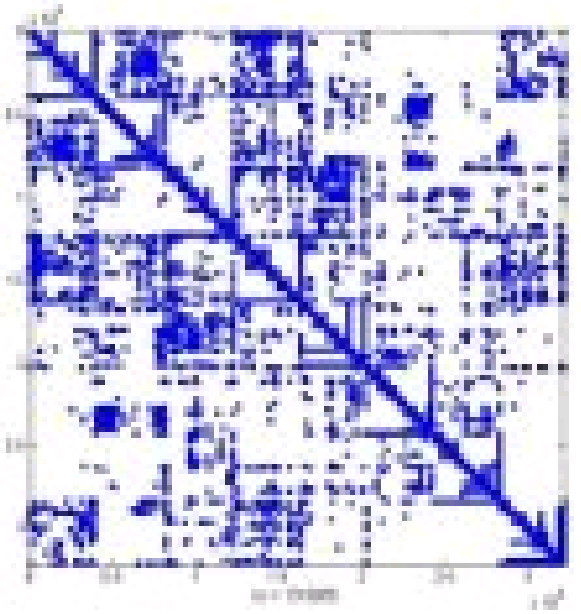


- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

- Performance may vary based on OS ver. and motherboard config.

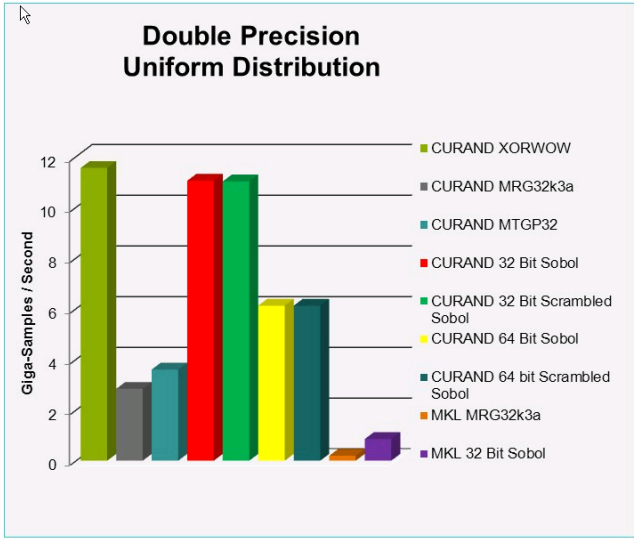
CuSPARSE

- Supports dense, COO, CSR, CSC, ELL/HYB and Blocked CSR sparse matrix formats
- Level 1 routines for sparse vector x dense vector operations
- Level 2 routines for sparse matrix x dense vector operations
- Level 3 routines for sparse matrix x multiple dense vectors (tall matrix)
- Routines for sparse matrix by sparse matrix addition and multiplication
- Conversion routines that allow conversion between different matrix formats
- Sparse Triangular Solve
- Tri-diagonal solver
- Incomplete factorization preconditioners ilu0 and ic0

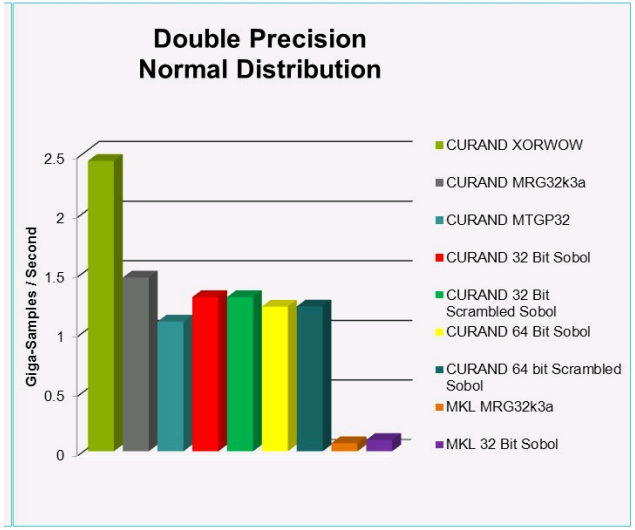


CuRAND

- **Flexible usage model**
  - Host API for generating random numbers in bulk on the GPU
  - Inline implementation allows use inside GPU functions/kernels, or in your host code
- **Four high-quality RNG algorithms**
  - MRG32k3a
  - MTGP Merseinne Twister
  - XORWOW pseudo-random generation
  - Sobol' quasi-random number generators, including support for scrambled and 64-bit RNG
- **Multiple RNG distribution options**
  - Uniform distribution
  - Normal distribution
  - Log-normal distribution
  - Single-precision or double-precision



- cuRAND 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz



•Performance may vary based on OS ver. and motherboard config.

CuRAND

1. **Create a generator:**

   curandCreateGenerator()

2. **Set a seed:**

   curandSetPseudoRandomGeneratorSeed()

3. **Generate the data from a distribution:**

   curandGenerateUniform()/curandGenerateUniformDouble() : Uniform

   curandGenerateNormal()/cuRandGenerateNormalDouble() : Gaussian

   curandGenerateLogNormal/curandGenerateLogNormalDouble() : Log-Normal

4. **Destroy the generator:**

   curandDestroyGenerator()

CuRAND

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

int main() {

int i, n = 100;
curandGenerator_t gen;
float *devData, *hostData;

// Allocate n floats on host
hostData = (float *) calloc (n,
 sizeof(float));

// Allocate nfloats on device
cudaMalloc((void **) &devData, n *
 sizeof(float));

// Create psudo-random number generator
curandCreateGenerator(&gen,
 CURAND_RNG_PSEUDO_DEFAULT);

// set seed
curandSetPseudoRandomGeneratorSeed(gen,
 1234ULL);
```

```
// generate n float on device
curandGenerateUniform(gen, devData, n);

// copy device memory to host
cudaMemcpy(hostData, devData, n *
 sizeof(float),
   cudaMemcpyDeviceToHost);


// show result
for (i = 0; i < n; i++) {
  printf("%1.4f ", hostData[i]);
}
printf("\n");

// Cleanup
curandDestroyGenerator(gen);

cudaFree(devData)
free(hostData)

return 0;

}
```

CuRAND

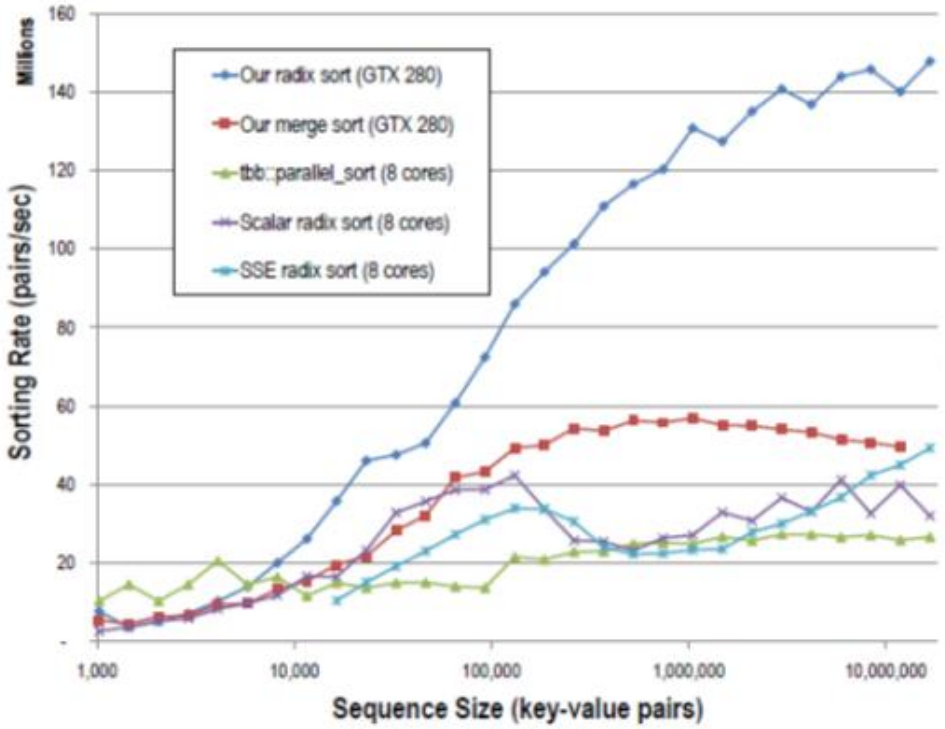
```
#include <stdio.h>
#include <stdlib.h>
#include <cuda.h>
#include <curand_kernel.h>

___global__ void
 setup_kernel(curandState *state)
{
  int id = threadIdx.x - blockIdx.x *
 64;
  // each thread gets same seed
  curand_init(1234, id, 0,
 &state[id]);
}

__global__ void generate_kernel(
  curandState *state, int *result)
{
  int id = threadIdx.x + blockIdx.x *
 64;
  int count = 0;
  unsigned int x;

  curandState localState = state[id];
```

```
  // generate pseudo-random unsigned
  for (int n = 0; n < 1000000; n++) {
    x = curand(&localState);
  }

  // copy state back to global memory
  state[id] = localState;

  // store results
  result[id] += count;
}
```

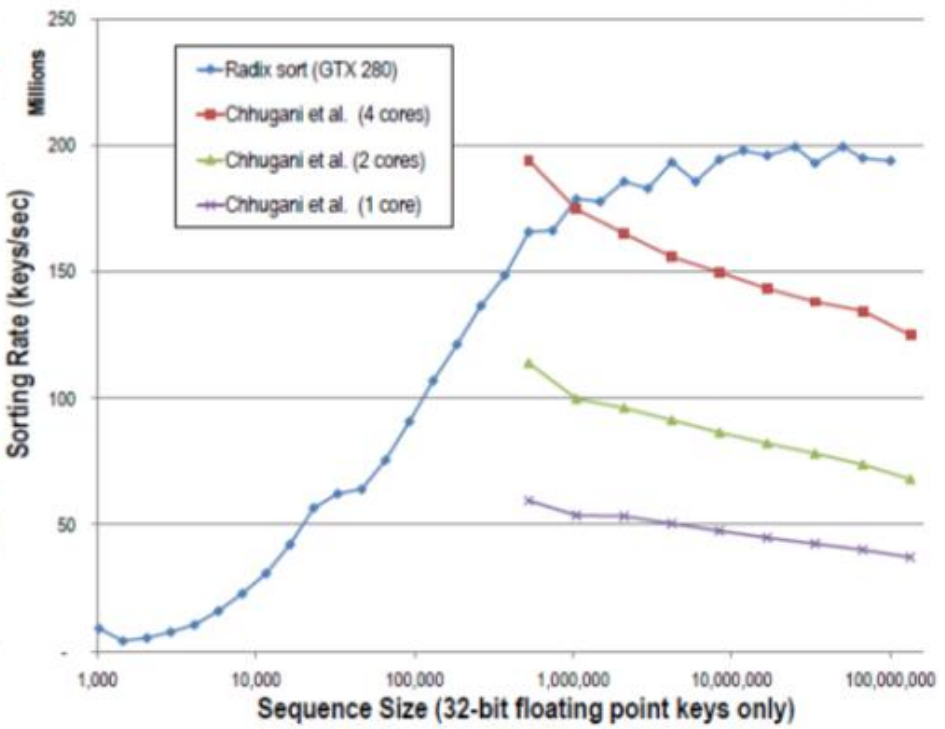
CUDPP 2.0

- CUDPP: CUDA Data Parallel Primitives library
- Libreria di algoritmi *data-parallel* per:
  - prefix-sum (“scan”)
  - parallel sort
  - reduction
- Operazioni complesse costruite sulle primitive:
  - hash table
  - array compaction
  - solutore di sistemi lineari tridiagonali
  - moltiplicazione matrice-sparsa/vettore
- Caratteristiche
  - Scritta in C/C++, disponibile API in C
  - Supporto per Windows, Linux e OSX
  - open source: http://code.google.com/p/cudpp/

CUDPP 2.0: prestazioni



(a) 8-core Clovertown (key-value pairs)



(b) 4-core Yorkfield (float keys only)

MAGMA: Matrix Algebra on GPU and Multicore Architectures

- La libreria LAPACK (Linear Algebra PACKage) è lo standard accettato per quanto riguarda accuratezza e interfaccia delle librerie che implementano operazioni complesse di algebra lineare
  - **Sono basate su BLAS**
- MAGMA è essenzialmente una re- implementazione di LAPACK su architetture eterogenee GPU + CPU multicore
  - **Sono basate su CUBLAS** e un sotto insieme di routine BLAS ottimizzate per GPU Fermi e Tesla sviluppate internamente al progetto MAGMA (`magma_blas`)
- MAGMA 1.x supporta il calcolo su più GPU CUDA enabled e CPU core (essenzialmente mediante versioni vendor multithreaded di BLAS e LAPACK
- É sviluppata da ICL group (Innovative Computing Laboratory) + collaboratori esterni + user community
- sono open source: http://icl.cs.utk.edu/projectsfiles/magma/
- Il layout dei dati segue la convenzione FORTRAN (column-major)
- É responsabilità dell'utente la gestione della memoria sul *device* e dei necessari trasferimenti dati

MAGMA: uso in C/C++

- MAGMA è sviluppata nativamente in C. Pertanto in ambiente C/C++ il suo utilizzo è immediato.
- l' interfaccia alla libreria è contenuta nel file:
  - `magma.h`
- è necessario allocare la memoria sul device usando il CUDA runtime

```
// Riduzione di una matrice simmetrica alla forma tridiagonale
// Interfaccia 'experimental'
#include <cuda.h>  // occorrono le funzioni per il memory management
#include <magma.h> // uso l' interfaccia GPU 'experimental'
// magma_int_t magma_dsytrd( char uplo, magma_int_t n, double *A,
//                 magma_int_t lda, double *d, double *e,
//                 double *tau, double *work, magma_int_t *lwork,
//                 double *da, double *dc, magma_int_t *info);
cudaError_t stat;
double *da, *dwork;
stat = cudaMalloc((void**)&da, n*n*sizeof(double));
stat = cudaMalloc((void**)&dwork, workSize* sizeof(double));
magma_dsytrd('U', n, A, lda, diagonal, offdiagonal, tau, work, lwork, da, dwork,
    &info)
```

MAGMA: uso in F90/2003

- MAGMA è sviluppata nativamente in C. Pertanto in ambiente F90/2003 il suo utilizzo richiede almeno la scrittura di `interface` e uso del modulo `ISO_C_BINDING`
- è necessario allocare la memoria sul device usando il CUDA runtime. Saranno necessarie interfaccie a `cudaMalloc/Free, cublasSetMatrix/cudaMemcpy`

```
!! Interfaccia nativa C:
!!  magma_int_t magma_dsytrd( char uplo, magma_int_t n, double *A,
!!                 magma_int_t lda, double *d, double *e,
!!                 double *tau, double *work, magma_int_t *lwork,
!!                 double *da, double *dc, magma_int_t *info);
!! Interfaccia F90/2003:
subroutine magma_dsytrd(uplo, n, a, lda, d, e, tau, work, lwork, da, dc, info)
  bind(C, name="magma_dsytrd")
  use iso_c_binding
  implicit none
  character, value:: uplo
  integer(C_INT), value :: n, lda
  integer(C_INT) :: info, lwork
  type(C_PTR), value :: a, d, e, tau, work, da, dc
  ! NB: type(C_PTR), value == void*
end subroutine magma_dsytrd
```

CUDA Thrust

A C++ template library for CUDA

- **Mimics the C++ STL**

- **Two containers**
  - **Manage memory on host and device:**

    thrust::host_vector<T>

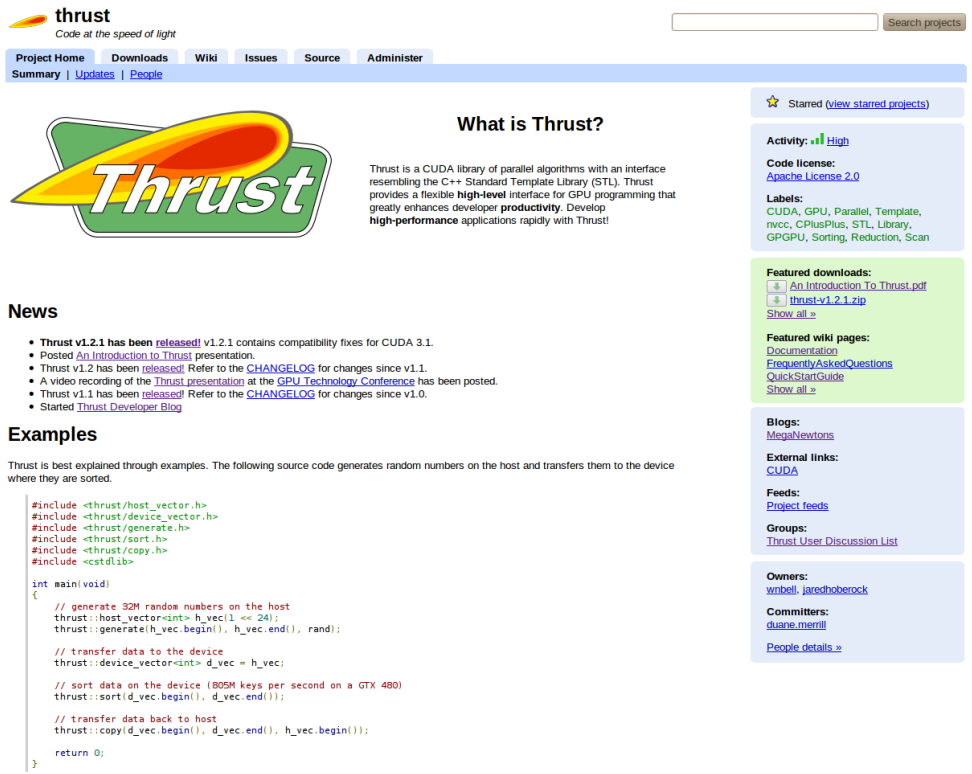
    thrust::device_vector<T>

- **Algorithms**
  - **Sorting, reduction, scan, etc:**

    thrust::sort()

    thrust::reduce()

    thrust::inclusive_scan()
  - **act on ranges of the container data by pair of iterators (a sort of pointers)**



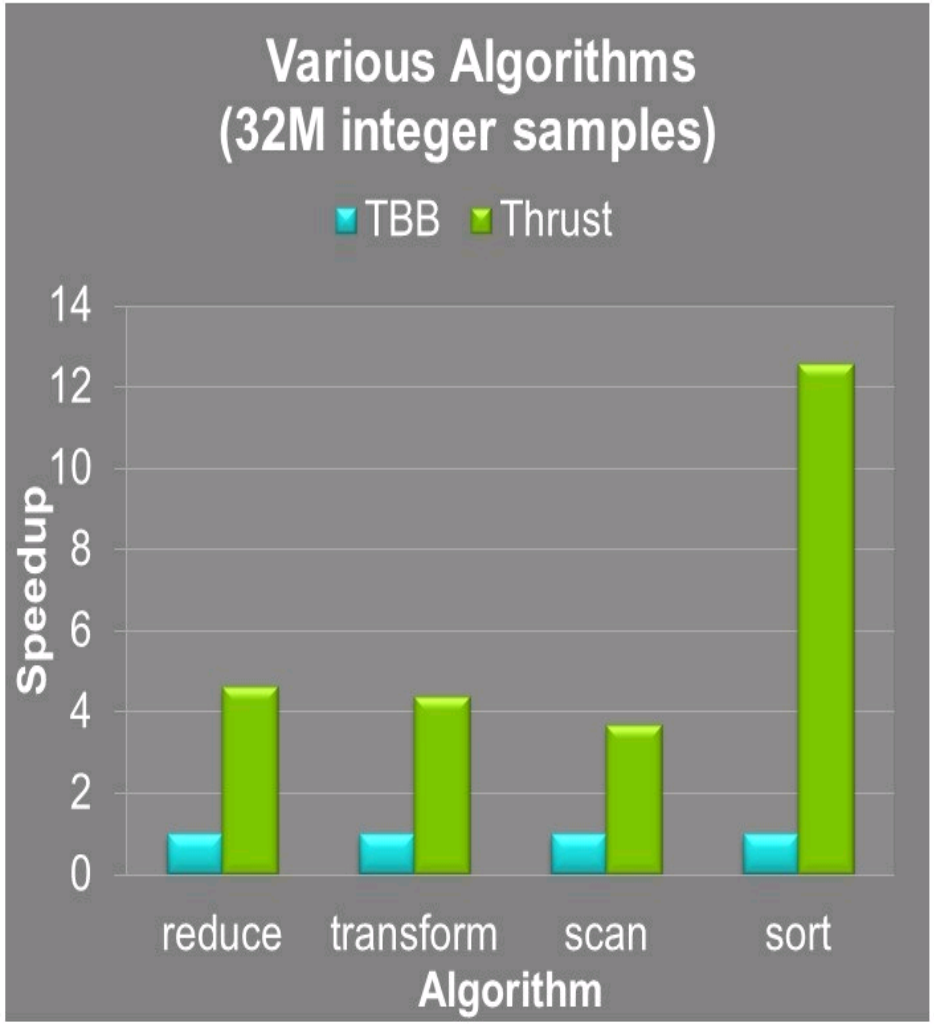
CUDA Thrust

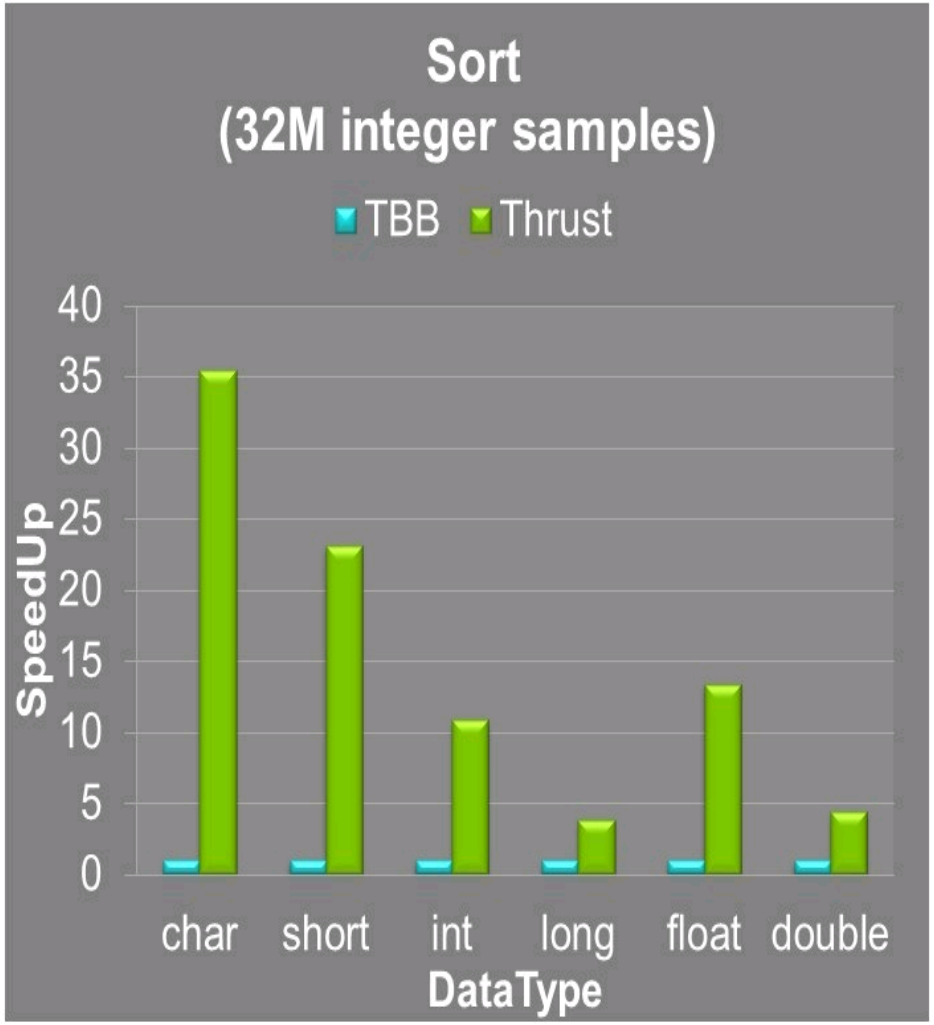
```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/sort.h>
#include <thrust/copy.h>
#include <cstdlib>

int main(void)
{
    // 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());

    return 0;
}
```

CUDA Thrust

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/generate.h>
#include <thrust/reduce.h>
#include <thrust/functional.h>
#include <cstdlib>

int main(void)
{
  // 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 x = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::plus<int>());
  return 0;
}
```

CUDA Thrust





- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

- Performance may vary based on OS ver. and motherboard config.

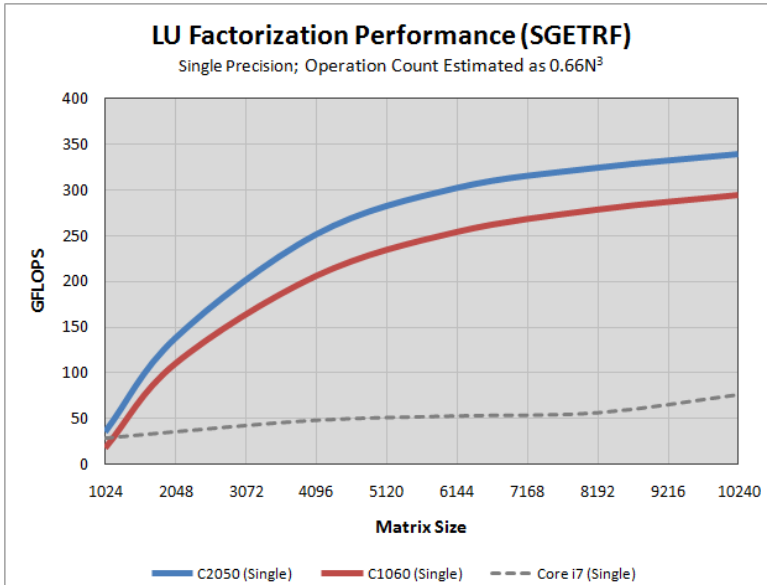
Lapack for CUDA: CULA Library

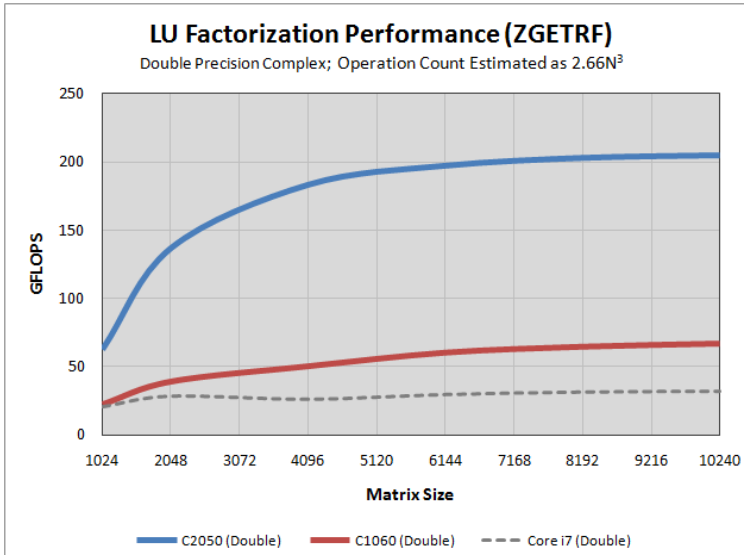
http://www.culatools.com

Proprietary library that implements the LAPACK in CUDA, which is available in several versions.

The speed-up of the picture on the right refers to:

CPU: Quad-core Intel Core i7 930 @ 2.8 GHZ CPU
GPU: NVIDIA Tesla C1060
GPU: NVIDIA Tesla C2050





Bibliografia

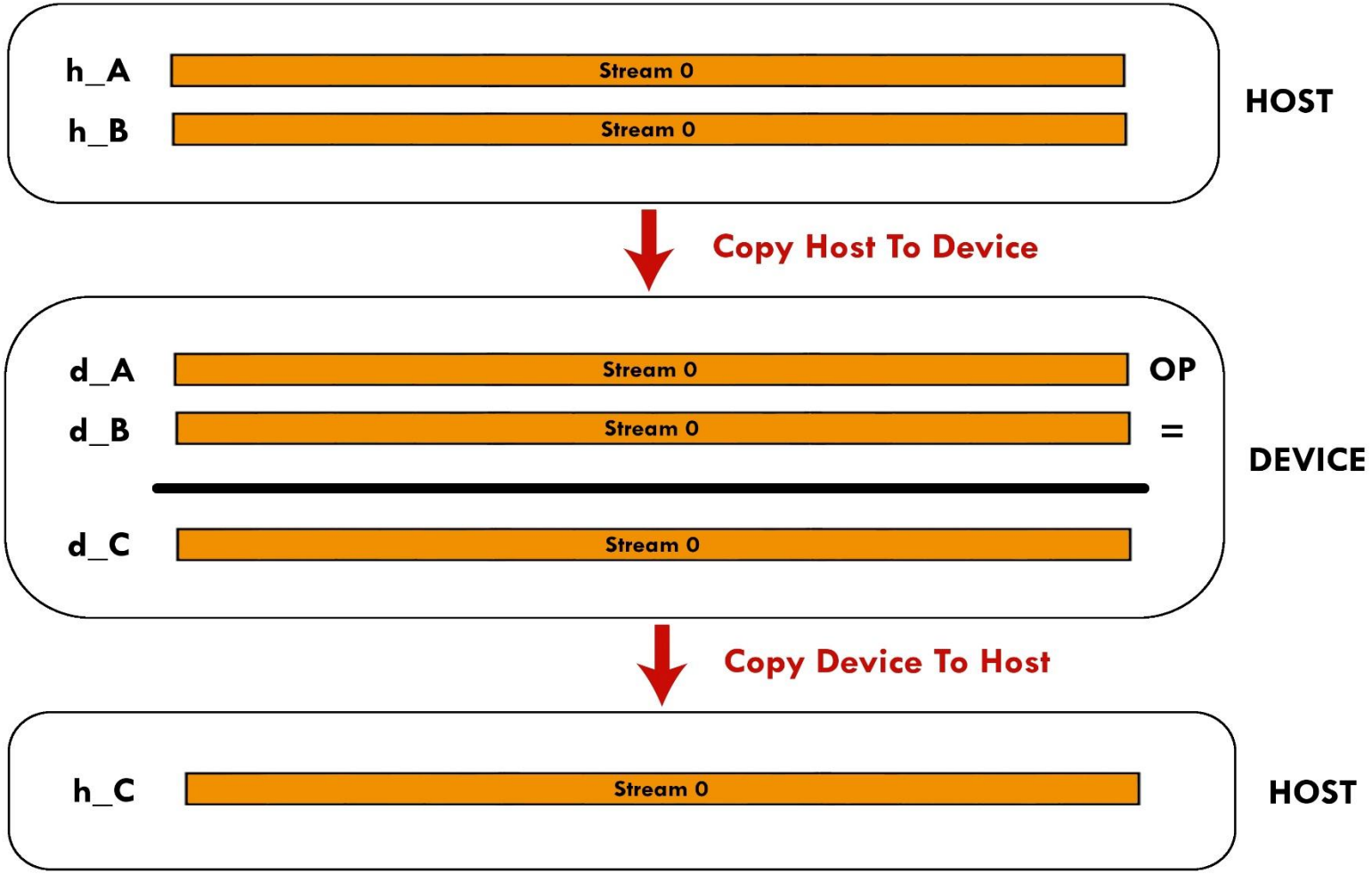
- ✓ CUDA C Programming Guide
- ✓ PGI CUDA fortran, http://www.pgroup.com/doc/pgicudaforug.pdf
- ✓ CUDA C Best Practices Guide
- ✓ NVIDIA CUDA Library Documentation (Doxygen –generated Reference Manual)
- ✓ Tuning CUDA Applications for Fermi
- ✓ Kirk and Hwu, Programming Massively Parallel Processors
- ✓ 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, Benchmarking GPUs to tune dense linear algebra
- ✓ V. Volkov, Better performance at lower occupancy, GPU Technology Conference 2010 (GTC 2010)
- ✓ J. Dongarra et al. "An Improved MAGMA GEMM for Fermi GPUs"

Esercizio 1: versione naive

Scrivere un programma in C o F90 che esegua i seguenti compiti.

- Allocare gli array di dati in singola precisione sull'*host*: `h_A, h_B, h_C` di dimensione `nSize`
- Inizializzare gli array `h_A` e `h_B` mediante la funzione `initArrayData()` in C o `RANDOM_NUMBER()` in F90
- Allocare gli array di dati in singola precisione sul *device*: `d_A, d_B, d_C`
- Trasferire gli array `h_A` e `h_B` su `d_A` e `d_B`, rispettivamente
- Lanciare il kernel `arrayFunc()` che elabora i dati contenuti in `d_A` e `d_B` e scrive il risultato su `d_C`
- Copiare `d_C` su `h_C`
- Misurare il tempo di calcolo mediante cudaEvent includendo anche i trasferimenti di memoria
- Eseguire la versione `funcArrayCPU()`
- Misurare il tempo di esecuzione tramite `cudaEvent`
- Calcolare lo Speed Up CPU/GPU

Esercizio 1: versione naive

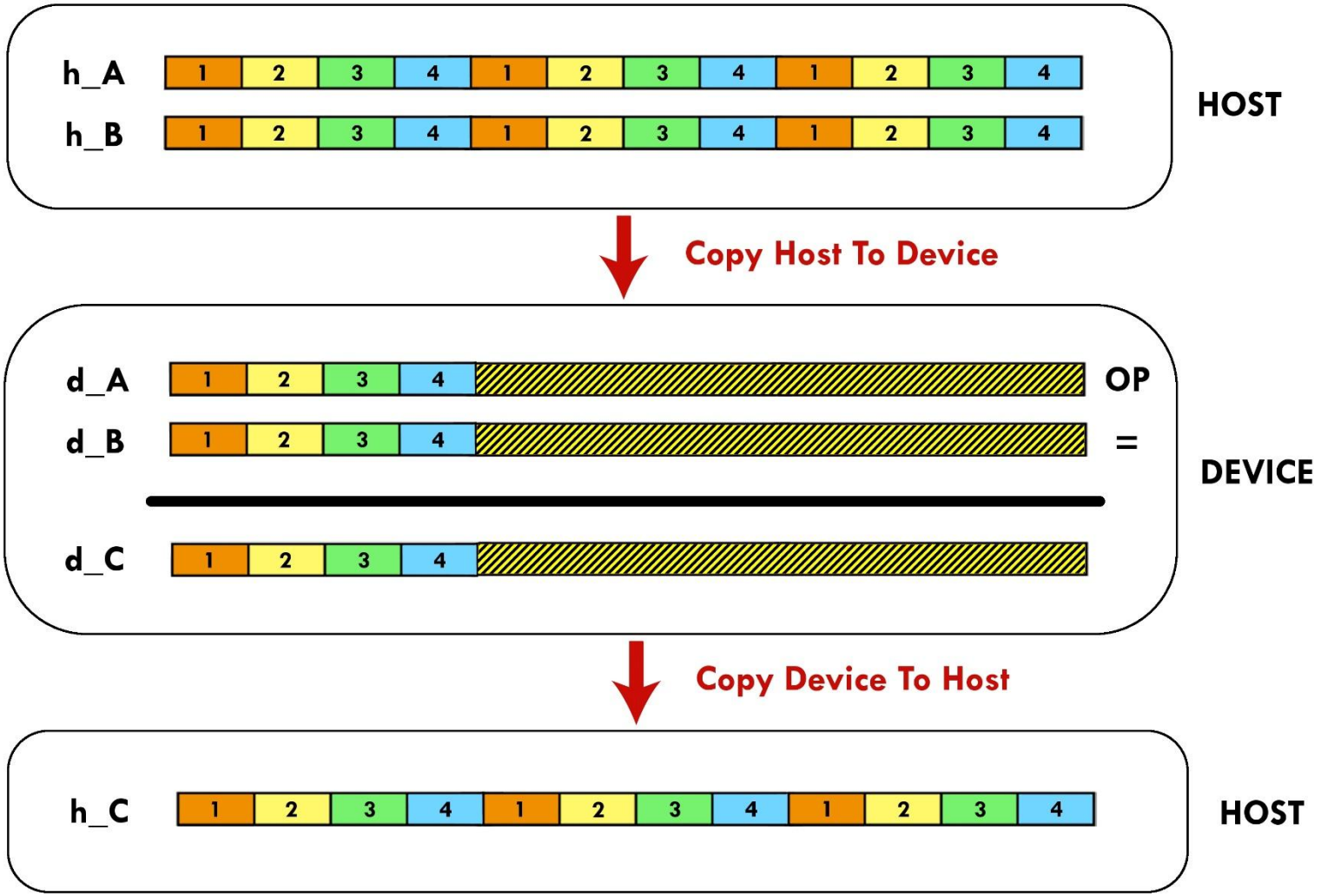


Esercizio 2: versione cudaStream

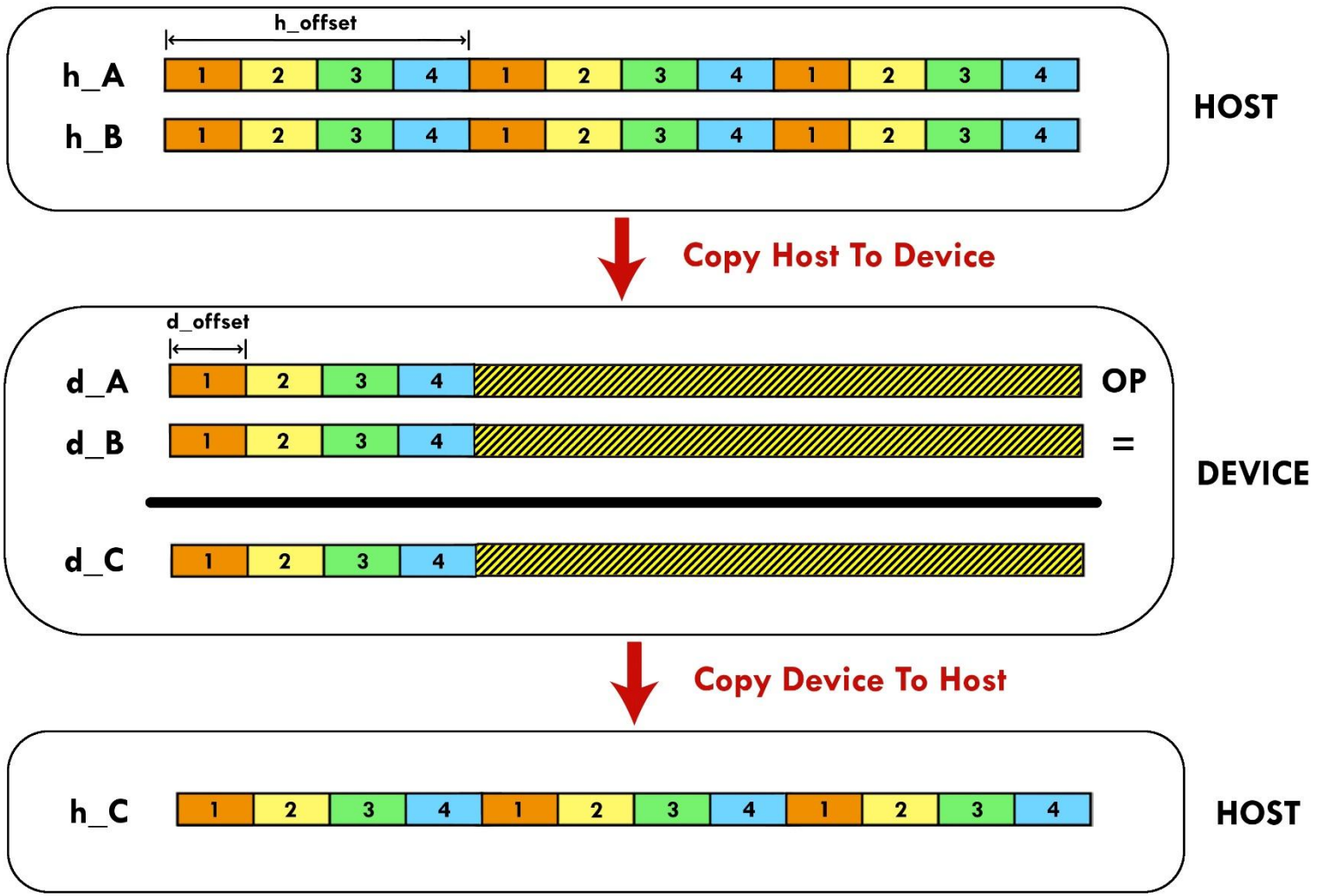
Scrivere un programma in C o F90 che esegua i seguenti compiti.

- Allocare gli array di dati sull'*host*: `h_A, h_B, h_C` di dimensione `nSize`
- Inizializzare gli array `h_A` e `h_B` utilizzando la funzione `initArrayData` in C o `RANDOM_NUMBER()` in F90
- Suddividere l'elaborazione di `h_A` e `h_B` in "chunk" di dimensione `chunk_size`
- Creare i cudaStream in numero pari a `streams_number`
- Allocare gli array di dati sul *device*: `d_A, d_B, d_C` dimensionati a `chunk_size * streams_number`
- Assegnare a ciascun cudaStream l' elaborazione di un certo numero di chunk. Ogni stream dovrà:
  - Copiare chunk di `h_A` e `h_B` da *host* a *device* in regioni oppurtune dei buffer `d_A` e `d_B`
  - Elaborare i chunk tramite il kernel `arrayFunc`
  - Copiare il risultato su `h_C`
- Misurare il tempo di esecuzione e valutare lo Speed Up rispetto all' esercizio precedente

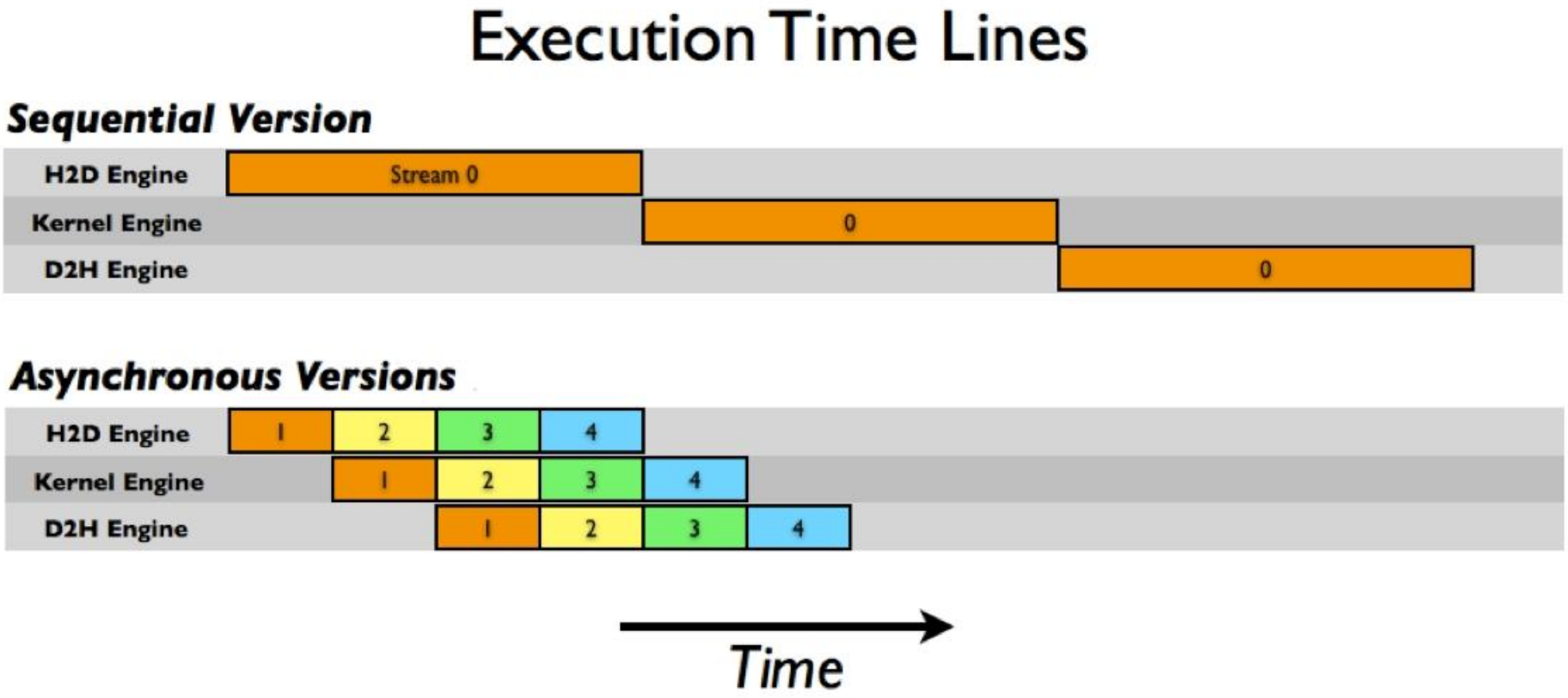
Esercizio 2: versione cudaStream



Esercizio 2: versione cudaStream



Esercizio 2: versione cudaStream



Esercizio 2: funzioni del CUDA runtime

Nuove funzioni del CUDA Runtime necessarie (C for CUDA):

- `cudaError_t cudaStreamCreate(cudaStream_t *stream)`
- `cudaError_t cudaStreamDestroy(cudaStream_t stream)`
- `cudaError_t cudaDeviceSynchronize(void)`
- `cudaErrot_t cudaMemcpyAsync(void* dst, void* src, size_t nbyte, enum cudaMemcpyKind kind,cudaStream_t stream)`

Nuove funzioni del CUDA Runtime necessarie (CUDA FORTRAN):

- `integer function cudaStreamCreate(stream)`
  `integer :: stream`
- `integer function cudaStreamDestroy(stream)`
  `integer :: stream`
- `integer function cudaDeviceSynchronize()`
- `integer function cudaMemcpyAsync(dst, src, nelements, kind, stream)`

Esercizio 3: Versione cudaStream Multi GPU

Scrivere un programma in C o F90 che esegua i seguenti compiti.

- Allocare gli array di dati sull'*host*: `h_A, h_B, h_C` di dimensione `nSize`
- Inizializzare gli array `h_A` e `h_B` utilizzando la funzione `initArrayData()` in C o `RANDOM_NUMBER()` in F90
- Suddividere l' elaborazione di `h_A e h_B` in "chunk" di dimensione `chunk_size`
- Assegnare a ciascuna GPU un uguale numero di chunk da eleborare
- Allocare gli array di dati sul *device*: `d_A, d_B, d_C` dimensionati a `chunk_size*streams_number`
- Creare i cudaStream in numero pari a `streamsNumber`
- Assegnare a ciascun cudaStream l' elaborazione di un certo numero di chunk. Ogni stream dovrà:
  - Copiare chunk di `h_A` e `h_B` da *host* a *device* in regioni oppurtune dei buffer `d_A` e `d_B`
  - Elaborare i chunk tramite il kernel `arrayFunc`
  - Copiare il risultato su `h_C`
- Misurare il tempo di esecuzione e valutare lo Speed Up rispetto all' esercizio precedente