GPGPU Evaluation – First experiences in Napoli

Silvio Pardi

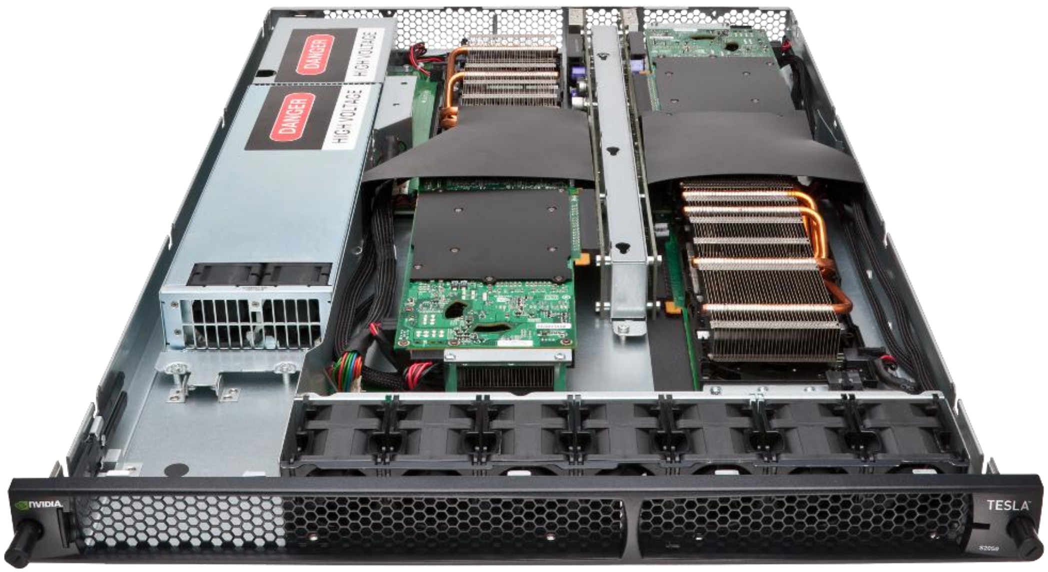
Goal of our preliminary tests

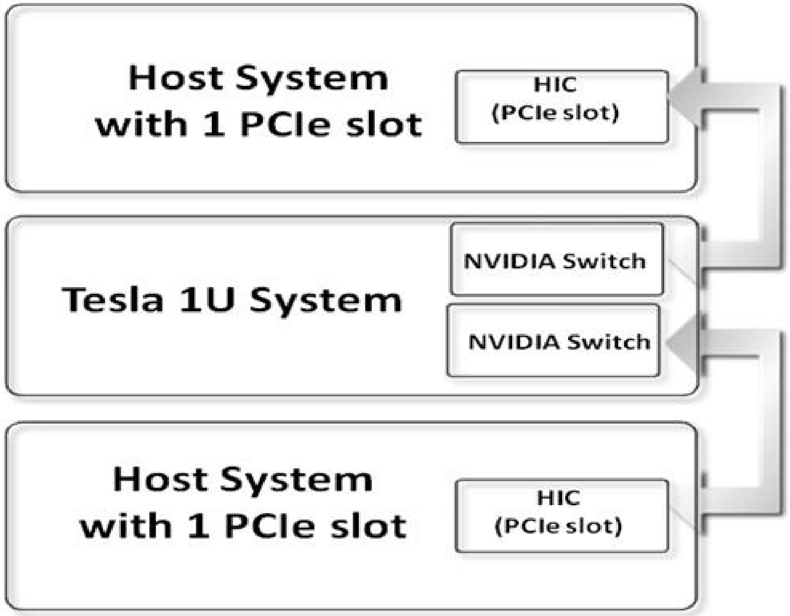
•Achieve know-how on GPGPU architectures in order to test the versatility and investigate the possible adoption for some specific tasks interesting for SuperB.

•multi-core Technology
•High speed and complex processing unit
• General Purpose

•many-core technology
• Hundred of simple Processing Units
• Designed to match the SIMD paradigm (Single Instruction Multiple Data)

The Hardware Available in Napoli





1U rack NVIDIA Tesla S2050

- 4 GPU Fermi
- Memory for GPU: 3.0 GB
- Core for GPU: 448
- Processor core clock: 1.15 GHz

2U rack Dell PowerEdge R510

- Intel Xeon E5506 eight-core @ 2.13 GHz
- 32.0 GB DDR3
- 8 hard disk SATA (7200 rpm), 500 GB

ENVIRONMENT

- Cuda compilation tools, release 4.0, V0.2.1221
- NVIDIA-Linux-x86_64-270.41.34.run
- cudatoolkit_4.0.17_linux_64_rhel5.5.run
- gpucomputingsdk_4.0.17_linux.run

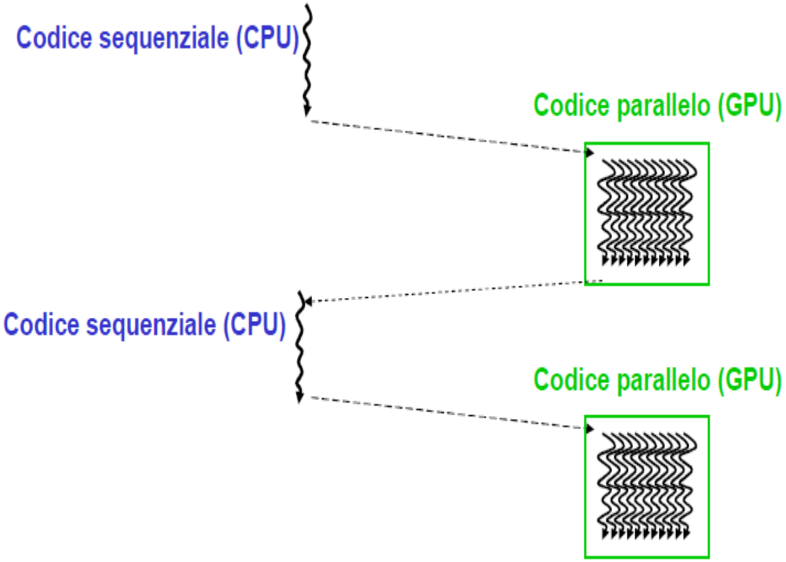
Tuning for lazily modality removal

The firsts test on the Tesla S2050 GPU with CUDA C show a starting overhead ~2 second due the "context" creation needed by the CUDA toolkit. The context is create **on demand** (**lazily**) and de-allocated when is not used.

SOLUTIONS (Suggested by Davide Rossetti):

1. Before the CUDA 4, Create a *dummy process always active that keeps alive the CUDA context "***nvidia-smi -l 60 -f /tmp/bu.log***"*
2. Since CUDA ver. 4 use the *-pm (persistence-mode) option from the nvidia-smi command to activate the GPU context.*

CUDA C

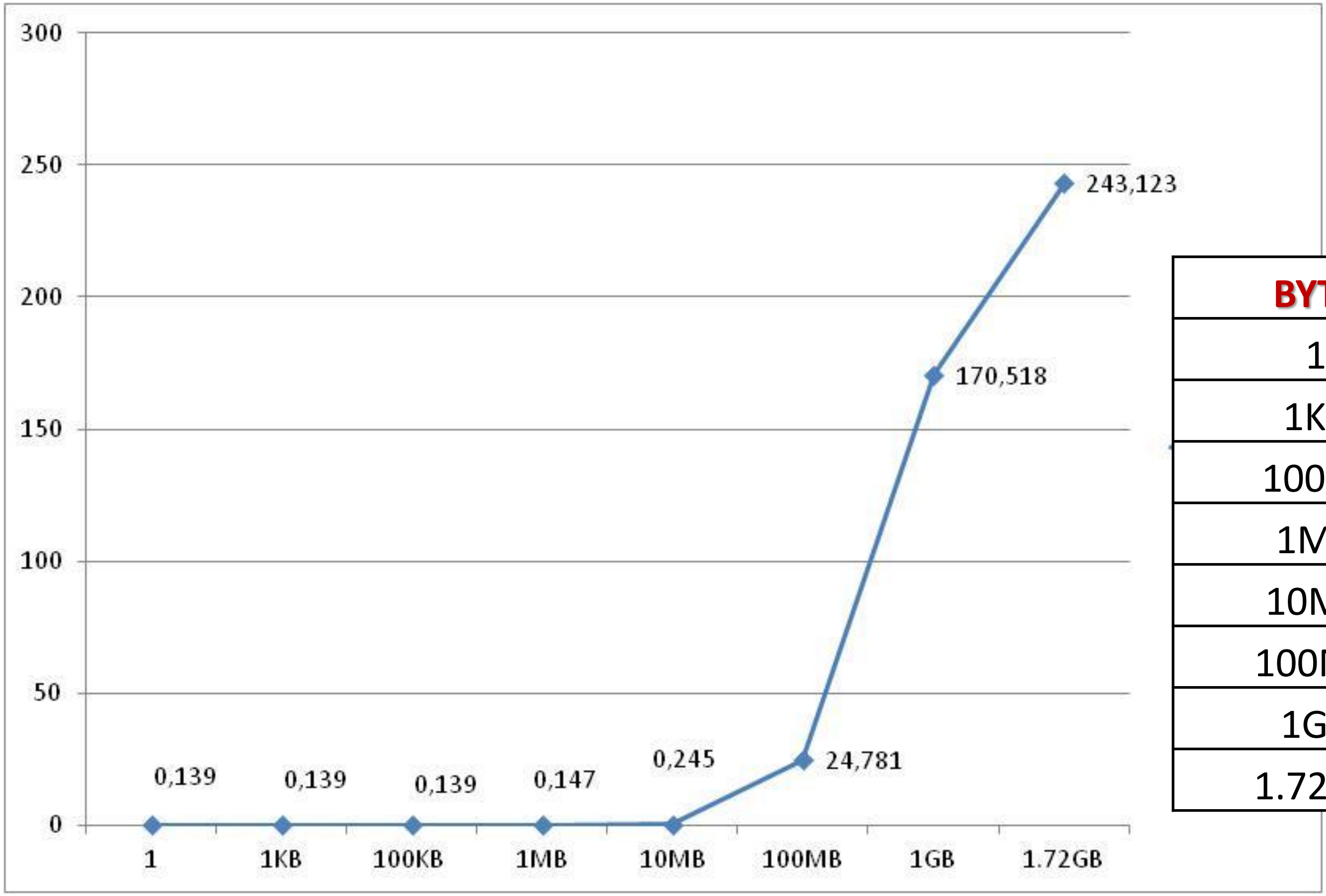


IBRID code: combination of standard C sequential code with parallel kernel in **CUDA C.**

Each code needs the following main steps:

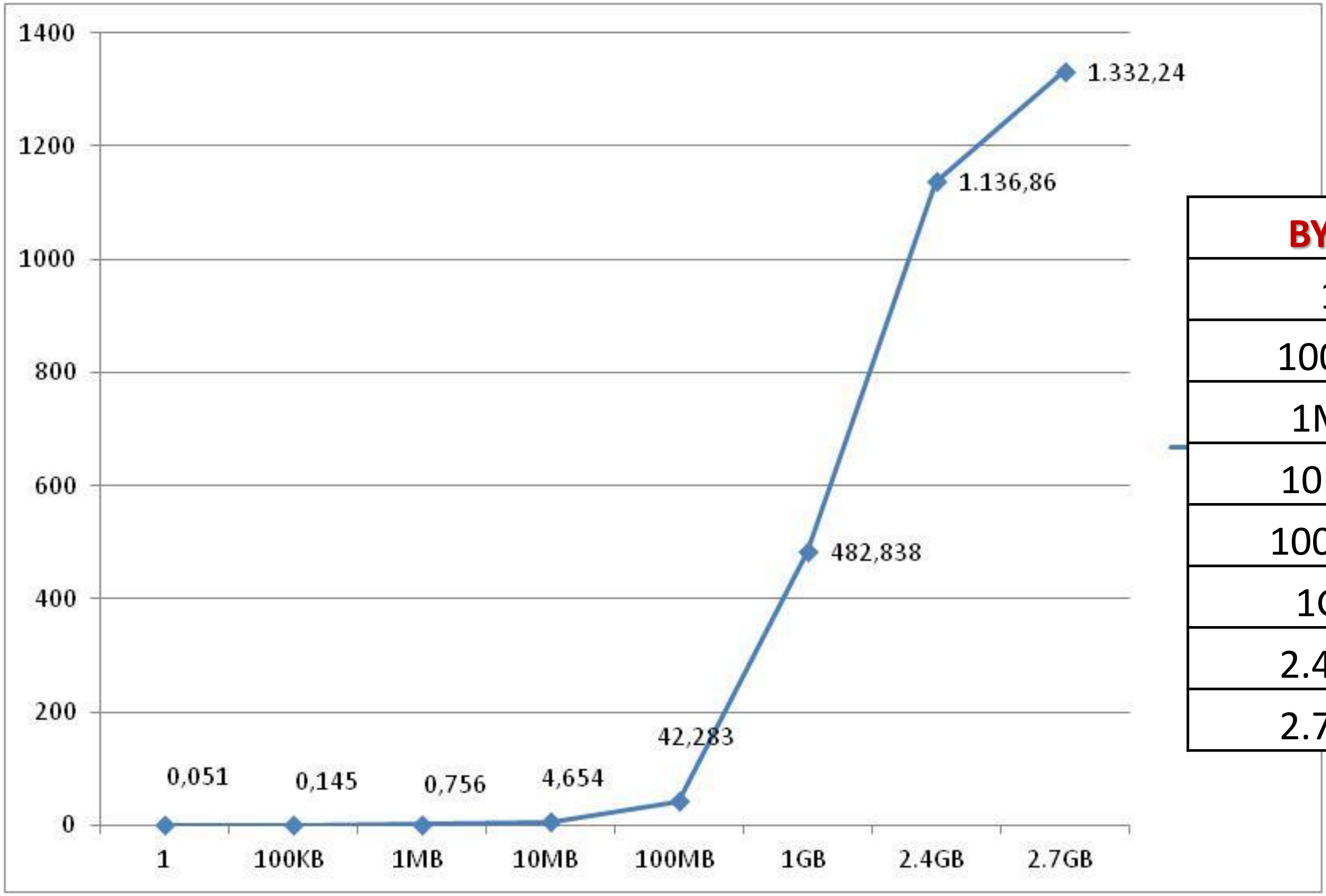
- ➢ GPU Memory Allocation (CudaMalloc)
- ➢ Data transfer between CPU and GPU (CudaMemCopy H2D)
- ➢ CUDA Kernel execution
- ➢ Data transfer between GPU and CPU (CudaMemCopy D2H)

CUDAMALLOC Benchmark



BYTE	TIME (ms)
1	0,139
1KB	0,139
100KB	0,139
1MB	0,147
10MB	0,245
100MB	24,781
1GB	170,518
1.72GB	243,123

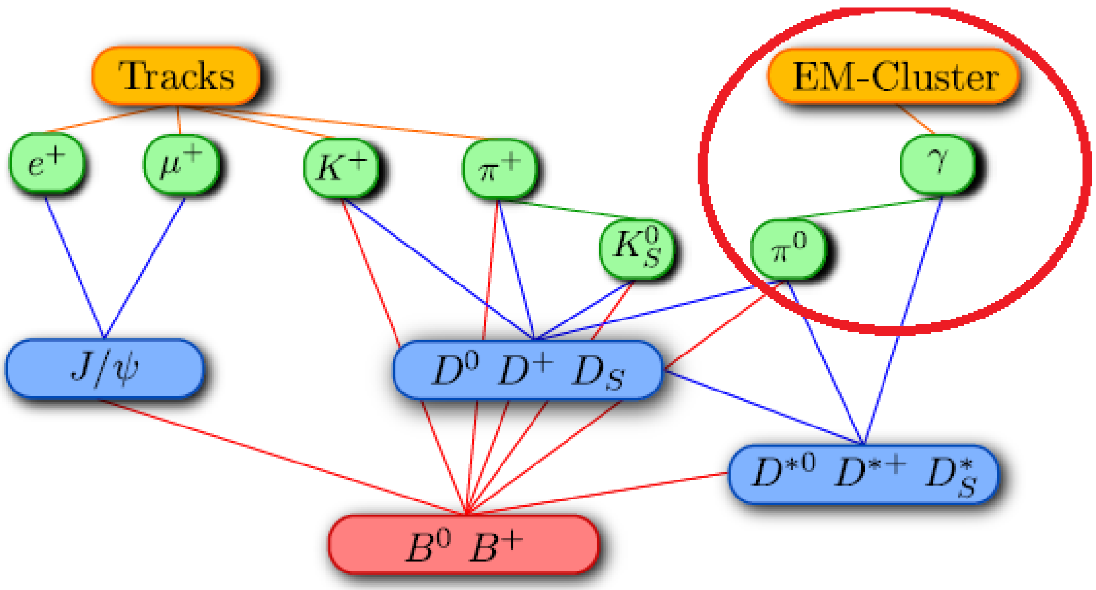
CUDAMEMCPY Benchmark



BYTE	TEMPO (ms)
1	0,051
100KB	0,145
1MB	0,756
10MB	4,654
100MB	42,283
1GB	482,838
2.4GB	1.136,858
2.7GB	1.332,238

Max bandwidth achieved : 16Gbit/s

Exercise: B-meson reconstruction like algorithm



stage	particelle
1	tracks, K_S, γ, π^0
2	$D^{\pm}_{(s)}$, D^0, e J/ψ
3	$D^{*\pm}_{(s)}$ e D^{*0}
4	$B^{\pm}$ e B^0

Combinatorial problem

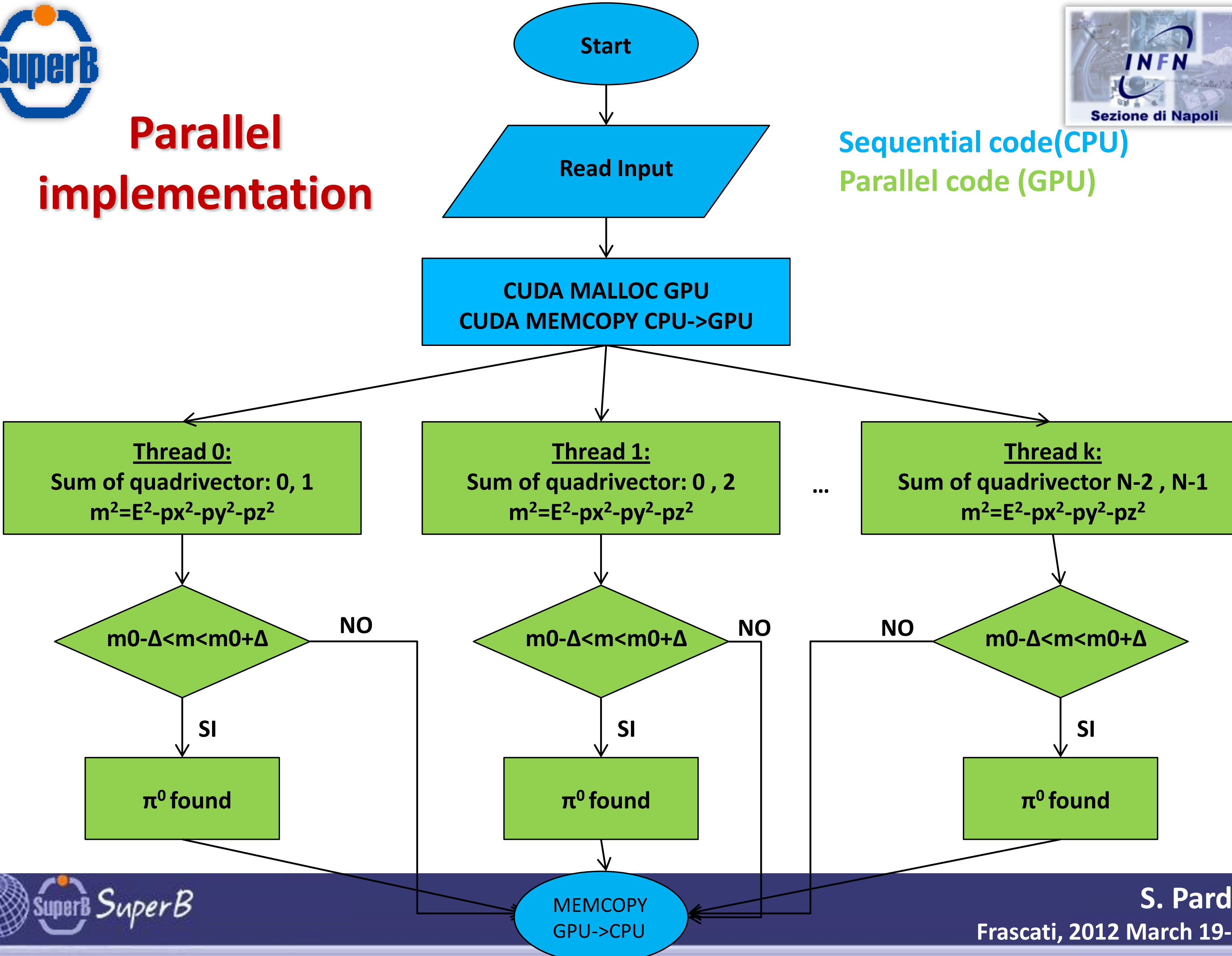
Problem Modellization: given N quadrivectors (spatial components and energy), combine all the couple without Repetition. Then calculate the mass of the new particle and check if the mass is in a range given by input.

GOAL: Understand the impact, benefits and limits of using the GPGPU architecture for this use case, through the help of a toy-model, in order to isolate part of the computation.

The algorithm has been implemented in C CUDA

Parallel implementation

Sequential code(CPU)
Parallel code (GPU)



Cuda implementation (1/2)

GPU Function

Memory Allocation on GPU

Data transfer from CPU to GPU

Run GPU Kernel

Data transfer back from GPU to CPU

GPU Memory Free

```
...
//Definizione di procedure
__global__ void kernel(QVECT *, PI_0 *, unsigned long int, unsigned long int, float, float, int *);
__device__ void createPI0(QVECT *, PI_0 *, unsigned long int, unsigned long int, float, float, int *);

int main(int argc, char **argv){
        ...
        //Allocazione memoria vettori host e device
        Gamma_Host=(QVECT*)malloc(N*sizeof(QVECT));
        cudaMalloc( (void**)&Gamma_Dev,N*sizeof(QVECT) );
        ...
        //Copio i dati dall'host al device
        cudaMemcpy(Gamma_Dev,Gamma_Host,N*sizeof(QVECT),cudaMemcpyHostToDevice);
        ...
        //Invoco la procedura kernel sul device
        kernel<<<dim3(1,1,1),dim3(190,1,1)>>>(Gamma_Dev,Pi0_Dev,N,coefBin,massa0,delta,dimPi0_d);
        cudaThreadSynchronize(); //Barriera di sincronizzazione thread
        ...
        //Copio i risultati dal device all'host
        cudaMemcpy(dimPi0_h,dimPi0_d,sizeof(int),cudaMemcpyDeviceToHost);
        Pi0_Host=(PI_0*)malloc(*dimPi0_h*sizeof(PI_0)); //Alloco memoria Pi 0 Host
        cudaMemcpy(Pi0_Host,Pi0_Dev,*dimPi0_h*sizeof(PI_0),cudaMemcpyDeviceToHost);
        ...
        //Rilascio memoria
        cudaFree(Gamma_Dev);
        cudaFree(Pi0_Dev);
        cudaFree(dimPi0_d);
        ...
}
```

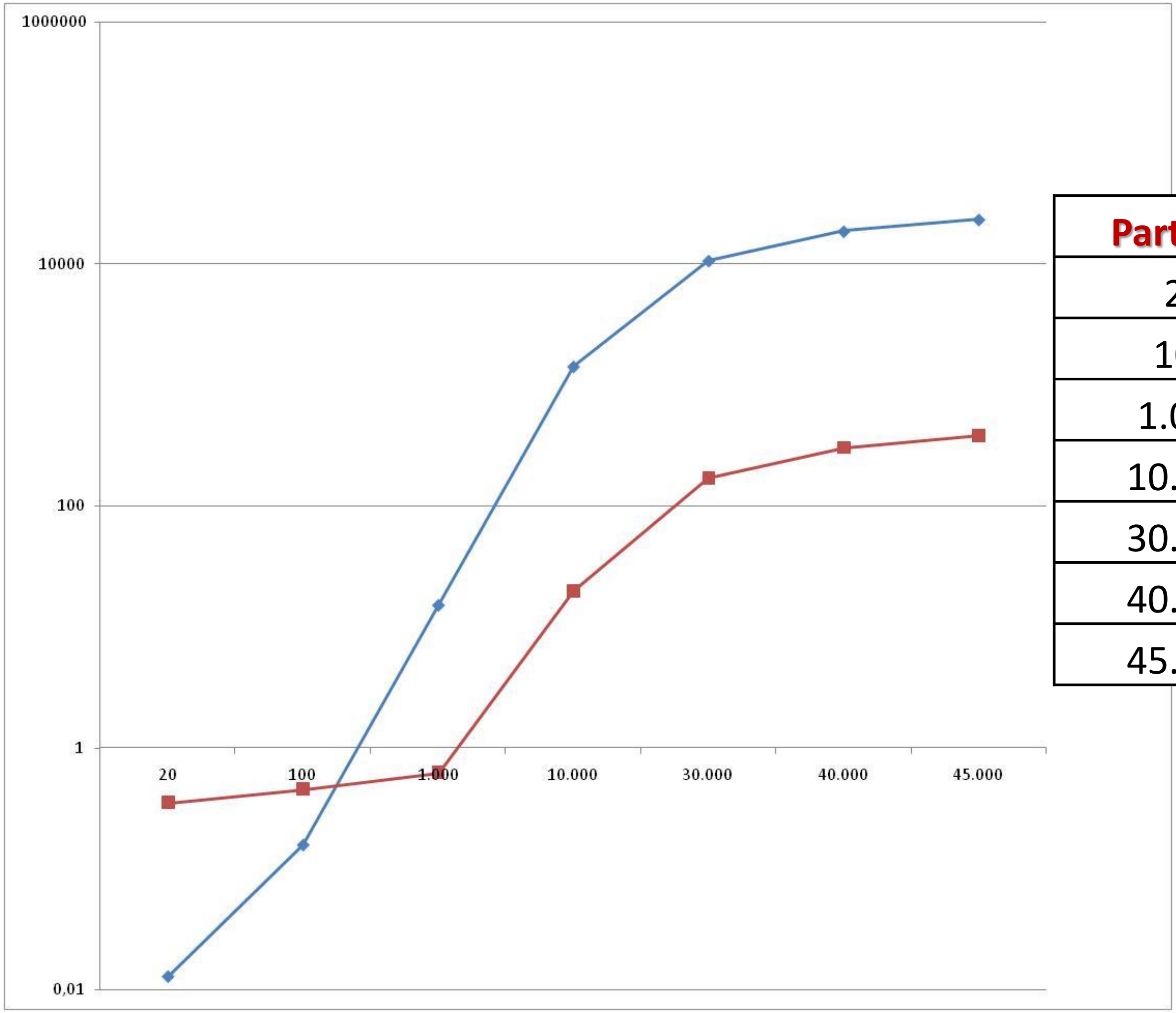
Cuda Kernel (2/2)

```
__global__ void kernel(QVECT *Gamma, PI_0 *Pi0, unsigned long int N, unsigned long int N2, float M0, float del, int *dimPi0){
        //Invoco procedura per il calcolo dei PI 0
        createPI0(Gamma,Pi0,N,N2,M0,del,dimPi0);
}

__device__ void createPI0(QVECT *Gamma, PI_0 *Pi0, unsigned long int N, unsigned long int N2, float M0, float del, int *dimPi0){
        ...
        //Dichiarazione variabile condivisa sul device
        __shared__ int Ind;
        //Il primo thread del blocco inizializza la variabile shared
        if (threadIdx.x == 0) {
           Ind = *dimPi0;
        }
        __syncthreads(); //Barriera di sincronizzazione threads
        //Calcolo del thread ID
        sh=threadIdx.x+blockDim.x*threadIdx.y;
        tid = blockDim.x*blockDim.y*(blockIdx.x + gridDim.x* blockIdx.y) + sh;
        if (tid<N2){
                ...
                //Somma di due quadrivettori
                Candidato.x=Gamma[i].x+Gamma[j].x;
                Candidato.y=Gamma[i].y+Gamma[j].y;
                Candidato.z=Gamma[i].z+Gamma[j].z;
                Candidato.Ene=Gamma[i].Ene+Gamma[j].Ene;
                //Controllo massa
                massa=(Candidato.Ene*Candidato.Ene)-(Candidato.x*Candidato.x)-(Candidato.y*Candidato.y)-(Candidato.z*Candidato.z);
                if(massa>M0-del && massa<M0+del){
                        loc=atomicAdd(&Ind,1);
                        Pi0[loc].x=Candidato.x;
                        Pi0[loc].y=Candidato.y;
                        Pi0[loc].z=Candidato.z;
                        Pi0[loc].Ene=Candidato.Ene;
                        Pi0[loc].g1=i;
                        Pi0[loc].g2=j;
                }
        }
        __syncthreads();
        //Il thread 0 di ogni blocco aggiorna la dimensione parziale dei PI 0
        if(threadIdx.x==0)
                atomicAdd(dimPi0,Ind);
}
```

Thread ID Computation

Quandrivector Composition

Testing Algorithm



Particles	Seq	Parallel
20	0,013 ms	0,351 ms
100	0,159 ms	0,457 ms
1.000	15,211 ms	0,621 ms
10.000	1.422,742 ms	19,699 ms
30.000	10.637,217 ms	170,23 ms
40.000	18.613,341 ms	301,756 ms
45.000	23.266,988 ms	380,989 ms

Configuration parameters
512 threads for block.
Advanced tuning can improve these results

Some Ideas

The first experience suggest to continue the investigation expecially in the following ways:

Tuning the memory management: Investigate the possibility to Overlapping Data Transfers and Computation through Async and Stream APIs.

Tuning the Grid/Block/Threads topology

Consider to rearrange the algorithms in term of operations per threads.

Conclusion

In Napoli we started to test the NVIDIA GPGPU architecture and we are investigating how to port HEP code on these architectures.

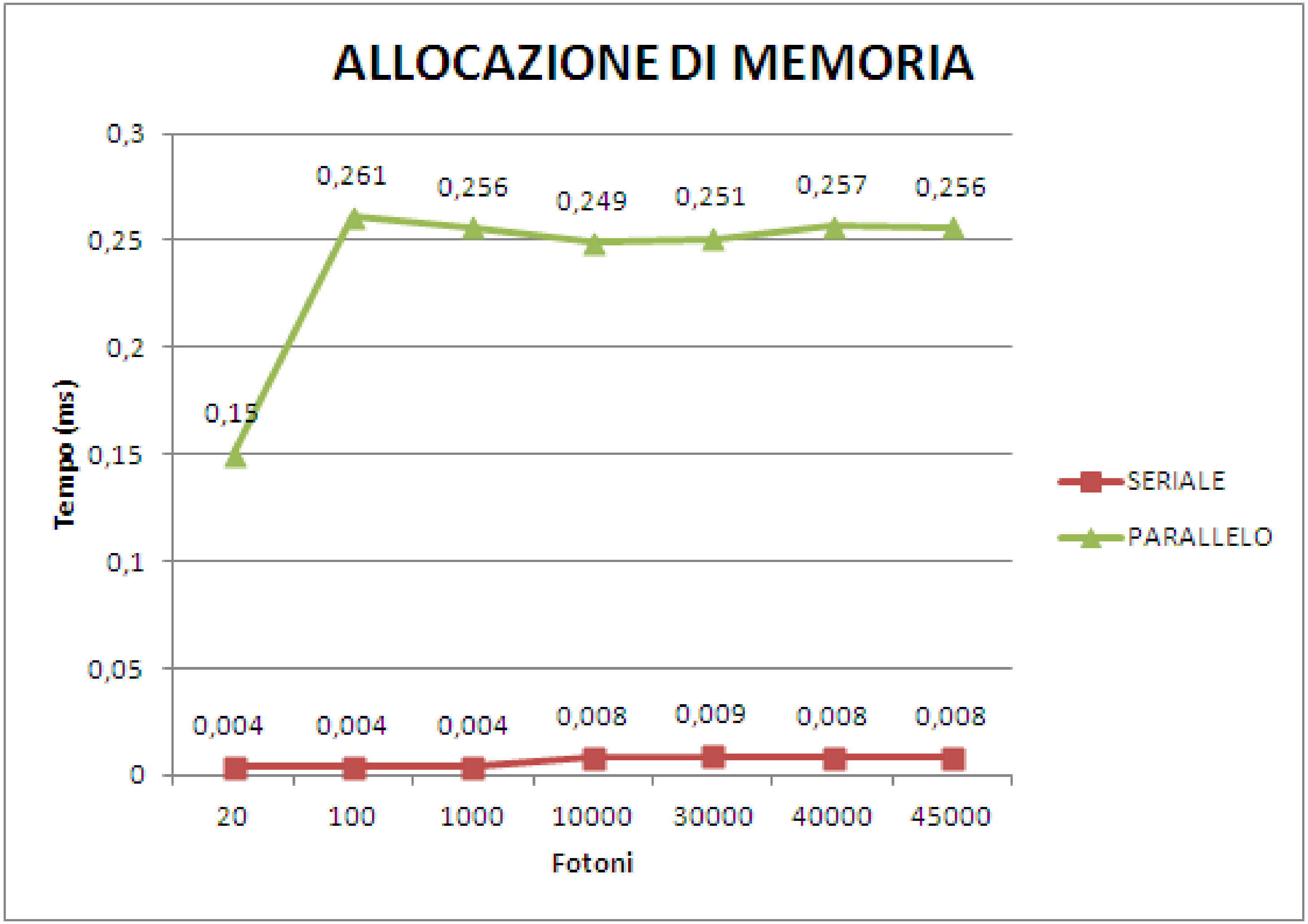
A first experience using a toy algorithm has show several aspects to take in account in GPU programming:

- Overhead management
- Memory management
- Algorithm re-engineerization
- Work distribution for each Thread
- Block and Thread topology definition

A lot of work is still due in order to achieve a full understanding of the architecture and the real benefits achievable. There are several work in progress.

END

Comparison malloc e cudaMalloc

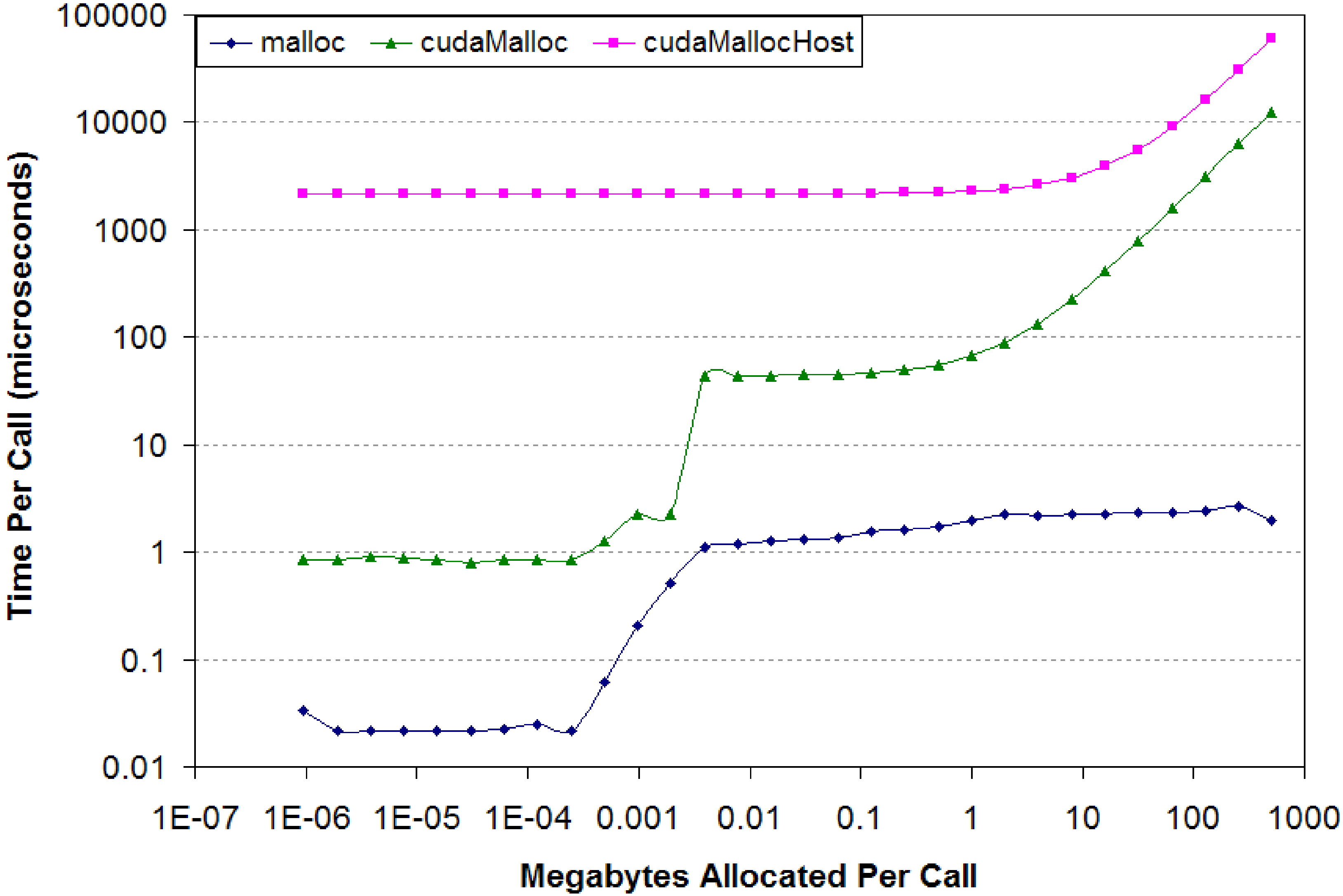


Algorithm for delegate a couple of vector to each Thread

```
...
//Calcolo del thread ID
sh=threadIdx.x+blockDim.x*threadIdx.y;
tid = blockDim.x*blockDim.y*(blockIdx.x + gridDim.x* blockIdx.y) + sh;

//Algoritmo per il calcolo degli indici giusti in base al thread
if (tid<N2){
        linIdx=N2-tid;
        i=int(N - 0.5 - sqrt(0.25 - 2 * (1 - linIdx)));
        z=(N+N-1-i)*i;
        j=tid - z/2 + 1 + i;

        if (i==j){
                i=i-1;
                j=N-1;
        }

        //Somma di due quadrivettori
        Candidato.x=Gamma[i].x+Gamma[j].x;
        Candidato.y=Gamma[i].y+Gamma[j].y;
        Candidato.z=Gamma[i].z+Gamma[j].z;
        Candidato.Ene=Gamma[i].Ene+Gamma[j].Ene;
        ...
```

Page-Locked Data Transfers

- `cudaMallocHost()` allows allocation of page-locked ("pinned") host memory
- Enables highest cudaMemcpy performance
  - 3.2 GB/s on PCI-e x16 Gen1
  - 5.2 GB/s on PCI-e x16 Gen2
- See the "bandwidthTest" CUDA SDK sample
- Use with caution!!
  - Allocating too much page-locked memory can reduce overall system performance
  - Test your systems and apps to learn their limits


malloc
cudaMalloc
cudaMallocHost
Time Per Call (microseconds)
100000
10000
1000
100
10
1
0.1
0.01
1E-07
1E-06
1E-05
1E-04
0.001
0.01
0.1
1
10
100
1000
Megabytes Allocated Per Call