Report RBSearch
Parallelizzata con MPI + OpenMP e CUDA + OpenMP

Autore: Antonio Sessa
Email:a.sessa108@studenti.unisa.it
Matricola: 0622702305
Professore: Francesco Moscato
fmoscato@unisa.it
Gennaio 2024

Contents

1 Descrizione del problema

Presentare una versione parallela dell'algoritmo di ricerca su un Red Black Tree con "OpenMP + MPI" e "Cuda + MPI" e compare gli algoritmi proposti con una soluzione nota con un singolo processing-node.

2 Soluzione proposta

2.1 Soluzione single processing-node

L'algoritmo di ricerca su un Red-Black Tree è identico all'algoritmo di ricerca per gli alberi binari di ricerca non bilanciati, ed è quindi la ricerca binaria. E' un algoritmo con complessità computazionale O(h) negli alberi binari di ricerca e di complessità o(log n) nei Red Black Tree per via della loro proprietà di essere bilanciati (i Red Black Tree hanno una altezza massima pari a 2*log(n+1)).

2.2 Approccio MPI

L'idea di base per l'approccio MPI è il seguente:

- Un processo, che possiamo denominare "master", si occupa dell'inizializzazione del Red Black Tree.
- Il processo master divide l'albero in N sottoalberi, dove N rappresenta il numero di processi disponibili.
- Ogni processo esegue la ricerca binaria sul suo sottoalbero.
- Una volta terminata la ricerca binaria su ogni sottoalbero, ogni processo invia al processo master i risultati della ricerca.
- Il processo master analizza i risultati ricevuti per determinare se il nodo è stato trovato; in caso affermativo, recupera il nodo.

Data che l'inizializzazione dell'albero prevede la lettura da un file dei valori, si potrebbe pensare di avere che ogni processo legga dal file a partire da un offset, e si crei così la propria parte di Red Black Tree. Una volta creata la propria parte di Red Black Tree, eseguire la ricerca ed inviare il nodo trovato (se trovato) al processo master. Il problema è che la struttura dati che rappresenta il nodo contiene i puntatori al figlio destro e al figlio sinistro, che diventerebbero non significativi nell'area di memoria del processo master e quindi questa soluzione non funzionerebbe.

2.3 Approccio Cuda

L'idea alla base per l'approccio CUDA è il seguente:

- L'host si occupa dell'inizializzazione del Red Black Tree
- L'host carica per intero l'albero sul dispositivo.
- Il dispositivo controlla ogni nodo dell'albero (ogni thread un nodo).
- Se il dispositivo trova il nodo contenente il dato cercato, lo restituisce all'host.

2.4 Linearizzazione dell'albero

La rappresentazione dell'albero utilizzato per i test non è contigua in memoria. Pertanto, sia per MPI che per CUDA, è stato necessario linearizzare l'albero. Questo perché il processo master non condivide la stessa area di memoria degli altri processi, e lo stesso vale per l'host e il device. La linearizzazione viene quindi fatta dal processo master / host sfruttando OpenMP. Per trasformare l'albero in un array ordinato, con l'ausilio di OpenMP si fanno due assunzioni:

- Si usano N thread, dove N è una potenza di due
- L'albero è pieno al livello log2(N)

2.5 Approccio OpenMP alla linearizzazione

L'idea alla base dell'approccio OpenMP con N Thread alla linearizzazione è il seguente:

- Gli N-1 nodi (sono N-1 perché assumiamo che il livello log2(N) sia pieno) che si trovano ai livelli prima di log2(N) vengono salvati in un array ordinato
- Si salvano gli N nodi che si trovano al livello log2(N) (pieno per assunzione) in un array ordinato.
- Si affida il nodo i-esimo dell'array dei nodi al livello log2(N) al thread i-esimo (identificati coi loro id).
- Conoscendo il numero di nodi che ogni thread visiterà (nella rappresentazione dei nodi è stata aggiunta una variabile che salva il numero di numero di nodi nel sottoalbero radicato in quel nodo) possiamo calcolarci dove ogni thread dovrà scrivere in un array condiviso, che rappresenta l'albero linearizzato e dove inserire gli N-1 nodi esclusi nell'array.
- Ogni thread esegue un traversamento in ordine dell'albero salvandosi i nodi nell'array condiviso.

In questo modo otteniamo un array ordinato che rappresenta il Red-Black Tree.

2.6 Il (non) approccio OpenMP alla ricerca binaria

OpenMP potrebbe essere usato anche all'interno della ricerca binaria eseguita da ogni processo nell'approccio MPI. Si è preferito però non implementare questa possibile soluzione, dopo aver analizzato i risultati del seguente report pubblicato dall'utente GitHub DVASAVDA : Link al Report. La conlcusioni a cui arriva il report, dopo l'analisi dei test effettuati, è che l'operazione di ricerca binaria non è sufficientemente costosa per benificiare della parallellizzazione.[1]. Inoltre va considerato, che testando tutto su una singola macchina l'overhead dovuta alla creazione dei thread sarebbe ancora maggiore (se impostiamo 4 omp thread e abbiamo 8 processi, verebbero genarati 32 thread che dovrebbero eseguire in parallelo sulla stessa cpu) .

3 Dati,Taglie,Ottimizzazioni

Ogni test case differisce per livello di compilazione, tipo di dato, e dimensione dell'albero. In particolare,

- testiamo per i livelli di compilazione -O0,-O1,-O2,-O3.
- testiamo per tre taglie diverse: 10000, 100000, 1000000

- per due tipi di dati: intero e array di interi di 100000 elementi[1]. I dati inseriti saranno casuali.

Testiamo per il tipo di dato array di 100000 elementi per avere un funzione di comparazione più onerosa. Per ogni test il valore cercato è il valore più grande. Ogni esecuzione dell'algoritmo CUDA ed MPI prevede anche la verifica con la versione seriale (questa non viene conteggiata nei tempi).

[1] L'array di interi di centomila elementi non verrà veramente caricato in memoria ma verrà solo usato per simulare una operazione di comparazione più lunga.

4 Setup Hardware

4.1 CPU

```
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
  Address sizes:         39 bits physical, 48 bits virtual
  Byte Order:            Little Endian
CPU(s):                  6
  On-line CPU(s) list:   0-5
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i5-8400 CPU @ 2.80
      GHz
    CPU family:          6
    Model:               158
    Thread(s) per core:  1
    Core(s) per socket:  6
    Socket(s):           1
    Stepping:            10
    BogoMIPS:            5616.00
    Flags:               fpu vme de pse tsc msr pae mce cx8
        apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr
        sse ss
                         e2 ss ht syscall nx pdpe1gb rdtscp lm
                            constant_tsc arch_perfmon rep_good
                            nopl xtopology cpuid p
                         ni pclmulqdq ssse3 fma cx16 pdcm pcid
                            sse4_1 sse4_2 movbe popcnt aes
                            xsave avx f16c rdrand hype
                         rvisor lahf_lm abm 3dnowprefetch
                            invpcid_single pti ssbd ibrs ibpb
                            stibp fsgsbase bmi1 avx2 sme
                         p bmi2 erms invpcid rdseed adx smap
                            clflushopt xsaveopt xsavec xgetbv1
                            xsaves md_clear flush_l1
                         d arch_capabilities
Virtualization features:
  Hypervisor vendor:     Microsoft
  Virtualization type:   full
Caches (sum of all):
  L1d:                   192 KiB (6 instances)
  L1i:                   192 KiB (6 instances)
  L2:                    1.5 MiB (6 instances)
  L3:                    9 MiB (1 instance)
```

4.2 RAM

```
Density:                16GB
```

4.3 GPU

```
CUDA Version:           12.3
GPU  Name:              NVIDIA GeForce RTX 3060 Ti
Memory Size:            8 GB
Memory Type:            GDDR6
Memory Bus:             256 bit
Bandwidth:              448.0 GB/s
L1 Cache:               128 KB (per SM)
L2 Cache:               4 MB
Compute Capability:     8.6
NVIDIA CUDA Cores:      4864
SM Count:               38
```

4.3.1 C.C 8.6

```
Blocks per SM:      16
Thread per SM:      1536
Registers per SM:   65536
```

4.3.2 Risultati del bandwidthTest

```
 Device 0: NVIDIA GeForce RTX 3060 Ti
 Quick Mode

 Host to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     12.5

 Device to Host Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     11.9

 Device to Device Bandwidth, 1 Device(s)
 PINNED Memory Transfers
   Transfer Size (Bytes)        Bandwidth(GB/s)
   32000000                     400.7
```

5 Risultati

I risultati riportati sono i valori minimi ricavati dopo 50 iterazioni. Il codice sorgente produce anche tabelle con i valori medii (ottenuti anche scartando eventuali valori troppo grandi / piccoli). Si è preferito riportare nel report i valori minimi per gestire la variabilità dei tempi (il programma può venire rallentato

da fattori esterni, quindi il run-time minimo è considerabile più veritiero). In particolare, i tempi minimi sono per ogni campo (questo spiega la discrepanza tra total time e gli altri campi). Il tipo di dato riportato nelle seguenti tabelle è array di 100000 elementi. Il codice sorgente produce anche tabelle per il tipo di dato intero, ma essendo la ricerca binaria su un albero di interi troppo veloce, viene sempre riportato 0.000 secondi come tempo di esecuzione. I campi controllati per MPI sono i seguenti:

- tempo totale
- tempo di comunicazione, il tempo impiegato per eseguire le funzioni di passaggio dati di MPI, in particolare sono una broadcast, una scatter e una gather
- tempo della ricerca binaria, misurata sul processo 0.[2]
- tempo di creazione dell'albero
- tempo di linearizzazione dell'albero

I campi controllati per CUDA sono i seguenti:

- tempo totale
- tempo GPU, ovvero le GPU Activities (Kernel + Memcpy)
- tempo esecuzione della intera funzione di ricerca con CUDA, ovvero GPU Activities + tree linearization + CUDA API Calls (cudaMallocHost + cudaLaunchKernel)
- tempo di creazione dell'albero
- tempo di linearizzazione dell'albero

I campi controllati per la versione seriale sono i seguenti:

- tempo totale
- tempo di ricerca
- tempo di creazione dell'albero

Per MPI verranno eseguiti i test per diverse configurazione di numero di processi, numero di thread OpenMP taglie o ottimizzazioni. Per Cuda verranno eseguiti i test per diversi numeri di thread CUDA, numero di thread OpenMP taglie o ottimizzazioni. Per CUDA abbiamo una occupancy (il rapporto tra il numero di warp attivi per multiprocessor e il numero massimo di warp attivi possibili) teorica del 100% per 256 e 512 thread,mentre per 1024 una occupancy teoretica del 66.67%[3] Per il seriale verrano eseguite i test per diverse taglie e ottimizzazioni.

[2]Per come è strutturato il programma il processo 0 non sarà mai quello che troverà l'elemento cercato

[3]Valori ottenuti con il tool NVIDIA Nsight Compute, visionabili nella cartella Nsights

5.1 Ottizzazione 0

5.1.1 Taglia 10000 MPI

	processes	omp$_t$*hreads*	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.022129	3.9e-05	0.016699	0.005032	0.000201
1	1	4	0.02237	4e-05	0.016706	0.004907	0.000323
2	1	8	0.022905	4.2e-05	0.016718	0.004995	0.000468
3	1	16	0.023117	4e-05	0.016734	0.004945	0.000993
4	2	1	0.022837	0.000315	0.017	0.005026	0.000201
5	2	4	0.023247	0.000239	0.017277	0.005041	0.000299
6	2	8	0.023593	0.000281	0.016972	0.005028	0.000415
7	2	16	0.02422	0.000237	0.017091	0.005049	0.000902
8	4	1	0.023269	0.000537	0.017029	0.005047	0.000199
9	4	4	0.023231	0.000471	0.01646	0.005018	0.000294
10	4	8	0.023982	0.000519	0.017	0.005034	0.00041
11	4	16	0.023683	0.000472	0.016858	0.005029	0.000617
12	6	1	0.030364	0.000725	0.021797	0.005038	0.0002
13	6	4	0.030225	0.000731	0.023085	0.005047	0.000292
14	6	8	0.030026	0.000829	0.022261	0.005011	0.000401
15	6	16	0.030337	0.000735	0.021825	0.005009	0.000612
16	8	1	0.03338	0.002266	0.022192	0.005042	0.000201
17	8	4	0.032741	0.002227	0.021687	0.005035	0.000287
18	8	8	0.032482	0.002194	0.019837	0.005027	0.000388
19	8	16	0.031665	0.002085	0.019649	0.005045	0.000617

5.1.2 Taglia 10000 CUDA

	threads	blocks	omp$_t$*hreads*	total$_t$*ime*	gpu$_t$*ime*	cuda search	tree$_c$*reation*	tree$_l$*inearizatio*
0	256	40	1	0.123596	0.000590	0.118553	0.005021	0.000106
1	256	40	4	0.123208	0.000514	0.118188	0.005021	0.000812
2	256	40	8	0.119854	0.000493	0.114792	0.005000	0.000500
3	256	40	16	0.120604	0.000482	0.115521	0.005062	0.000667
4	512	20	1	0.122312	0.000571	0.117188	0.005062	0.000167
5	512	20	4	0.122122	0.000511	0.117061	0.005020	0.000735
6	512	20	8	0.119250	0.000487	0.114208	0.005021	0.000729
7	512	20	16	0.120458	0.000505	0.115354	0.005083	0.000604
8	1024	10	1	0.123277	0.000566	0.118106	0.005170	0.000128
9	1024	10	4	0.122857	0.000517	0.117837	0.005020	0.000673
10	1024	10	8	0.119511	0.000504	0.114447	0.005043	0.000766
11	1024	10	16	0.119184	0.000507	0.114102	0.005020	0.000816

5.1.3 Taglia 10000 Seriale

	0
total$_t ime$	0.012
binary$_s earch_t ime$	0.007
tree$_c reation_t ime$	0.0

5.1.4 Taglia 100000 MPI

	processes	omp$_t hreads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.089467	0.000275	0.021137	0.063039	0.002886
1	1	4	0.087769	0.000274	0.020956	0.062735	0.001867
2	1	8	0.08743	0.000264	0.020999	0.062638	0.00185
3	1	16	0.088366	0.000263	0.020946	0.062767	0.002262
4	2	1	0.093009	0.000849	0.024808	0.062422	0.002851
5	2	4	0.091358	0.000762	0.024083	0.062161	0.001893
6	2	8	0.09269	0.000728	0.024464	0.062877	0.0019
7	2	16	0.09226	0.000619	0.024076	0.062274	0.002484
8	4	1	0.093937	0.001525	0.024301	0.062801	0.002778
9	4	4	0.093528	0.00174	0.024757	0.062724	0.001862
10	4	8	0.094259	0.001624	0.02413	0.062513	0.00187
11	4	16	0.094876	0.001575	0.024309	0.062595	0.002404
12	6	1	0.099845	0.002505	0.024761	0.063504	0.002826
13	6	4	0.097468	0.002433	0.026144	0.063212	0.001764
14	6	8	0.09625	0.00196	0.025316	0.062449	0.001826
15	6	16	0.098316	0.001968	0.026972	0.063235	0.002318
16	8	1	0.108233	0.004028	0.033198	0.062292	0.00272
17	8	4	0.106123	0.003056	0.032568	0.063823	0.001786
18	8	8	0.108054	0.00302	0.029146	0.063203	0.001893
19	8	16	0.10687	0.003245	0.033275	0.062807	0.00236

5.1.5 Taglia 100000 CUDA

	threads	blocks	omp$_t hreads$	total$_t ime$	gpu$_t ime$	cuda search	tree$_c reation$	tree$_l inearizatio$
0	256	391	1	0.192641	0.000739	0.120846	0.070205	0.003538
1	256	391	4	0.191844	0.000653	0.119756	0.070356	0.001889
2	256	391	8	0.187375	0.000712	0.115438	0.070188	0.001479
3	256	391	16	0.186818	0.000703	0.115614	0.069545	0.001568
4	512	196	1	0.193553	0.000726	0.121319	0.070468	0.003574
5	512	196	4	0.191208	0.000673	0.118937	0.070396	0.001938
6	512	196	8	0.187622	0.000713	0.115489	0.070289	0.001578
7	512	196	16	0.187673	0.000723	0.115755	0.070102	0.001633
8	1024	98	1	0.192467	0.000728	0.121311	0.069400	0.003600
9	1024	98	4	0.189723	0.000661	0.118511	0.069447	0.001809
10	1024	98	8	0.187796	0.000714	0.116082	0.069980	0.001592
11	1024	98	16	0.187545	0.000695	0.116705	0.069159	0.001636

5.1.6 Taglia 100000 Seriale

	0
total_time	0.078
$\text{binary}_search_time$	0.015
$\text{tree}_creation_time$	0.047

5.1.7 Taglia 1000000 MPI

	processes	$\text{omp}_threads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	1.305936	0.001253	0.025298	1.147335	0.08483
1	1	4	1.255955	0.001222	0.025164	1.151816	0.031545
2	1	8	1.245967	0.001229	0.025238	1.149359	0.022939
3	1	16	1.252443	0.001256	0.025271	1.155421	0.022842
4	2	1	1.314629	0.002216	0.029627	1.148709	0.08451
5	2	4	1.265835	0.002383	0.031137	1.154299	0.031404
6	2	8	1.257536	0.002225	0.030289	1.153757	0.022912
7	2	16	1.251524	0.002227	0.030048	1.149723	0.022135
8	4	1	1.323225	0.003403	0.036059	1.151325	0.085073
9	4	4	1.265177	0.003783	0.034649	1.147832	0.03155
10	4	8	1.267258	0.003782	0.036293	1.156876	0.022898
11	4	16	1.260829	0.00387	0.033679	1.151363	0.022634
12	6	1	1.326804	0.004755	0.032263	1.155251	0.084994
13	6	4	1.277252	0.004963	0.031239	1.156982	0.031587
14	6	8	1.264781	0.005617	0.03192	1.153851	0.022854
15	6	16	1.262035	0.004716	0.032598	1.149479	0.022042
16	8	1	1.344817	0.010962	0.037002	1.152235	0.084608
17	8	4	1.286267	0.010451	0.03651	1.155691	0.031737
18	8	8	1.277235	0.011354	0.03893	1.153588	0.023018
19	8	16	1.283692	0.012271	0.03686	1.157998	0.023001

5.1.8 Taglia 1000000 CUDA

	threads	blocks	$\text{omp}_threads$	total_time	gpu_time	cuda search	$\text{tree}_creation$	$\text{tree}_linearization$
0	256	3907	1	1.533851	0.002033	0.220340	1.269851	0.091319
1	256	3907	4	1.485120	0.001879	0.163060	1.277700	0.033720
2	256	3907	8	1.468918	0.002000	0.154694	1.270245	0.025980
3	256	3907	16	1.474500	0.002003	0.152348	1.278565	0.024761
4	512	1954	1	1.542477	0.002038	0.222295	1.275909	0.092818
5	512	1954	4	1.482979	0.001872	0.162333	1.276750	0.034479
6	512	1954	8	1.476174	0.002016	0.152130	1.280239	0.025826
7	512	1954	16	1.477256	0.002056	0.151326	1.281907	0.023977
8	1024	977	1	1.542778	0.002062	0.222689	1.275267	0.092622
9	1024	977	4	1.487468	0.001878	0.162660	1.280745	0.033362
10	1024	977	8	1.488333	0.002037	0.153689	1.290467	0.025778
11	1024	977	16	1.474600	0.002064	0.152644	1.276933	0.025000

5.1.9 Taglia 1000000 Seriale

	0
total_time	1.125
$\text{binary}_search_time$	0.015
$\text{tree}_creation_time$	1.093

5.2 Considerazioni, Speedup ed Efficiency

Come ci si poteva aspettare, anche dopo aver analizzando il report sulla parallelizzazione della ricerca binaria [1], l'algoritmo che ottiene prestazioni migliori è quello seriale, questo è vero per qualsiasi taglia. Essendo la ricerca su un albero bilanciato poco costosa computazionalmente, i benefici della parallelizzazione vengono superati dai deficit (come gli overhead di comunicazione, thread spawning, l'aumento delle chiamate di funzione). Anche non considerando la necessità di linearizzare l'albero, che potrebbe essere evitata con una rappresentazione contigua in memoria, difficilmente si possono ottenere prestazioni migliori della versione seriale.

Per i test con MPI dobbiamo anche tenere in considerazione che nei test lo usiamo su una singola macchina e non su un cluster per come è pensato, essendo un protocollo di comunicazione.

Per CUDA, pure se la funzione kernel è computazionalmente meno costosa, rimane l'overhead di dover spostare l'albero dalla memoria dell'host a quella del device. Inoltre non sembra esserci differenza tra l'utilizzo di blocchi di 256/512/1024 thread.

C'è anche da tenere in conto che i test riportati sono con dei dati che hanno una operazione di comparazione artificialmente resa più lunga.

In contrasto, prendiamo in considerazione l'operazione di linearizzazione, parallelizzata con OpenMP. Per la prima taglia, l'overhead della parallelizzazione non produce alcuno speep-up. Per la seconda taglia abbiamo uno speed-up significativo che è ancora migliore per la terza, che ci suggerisce un rapporto proporzionale tra la taglia e lo speedup.

In particolare, per la taglia pari a 100000 abbiamo lo speed-up maggiore con 4 thread[4]:

$$Speedup = 0.0027/0.0018 \approx 150\%$$

$$Efficienza = (0.0027/0.0018)/4 \approx 37\%$$

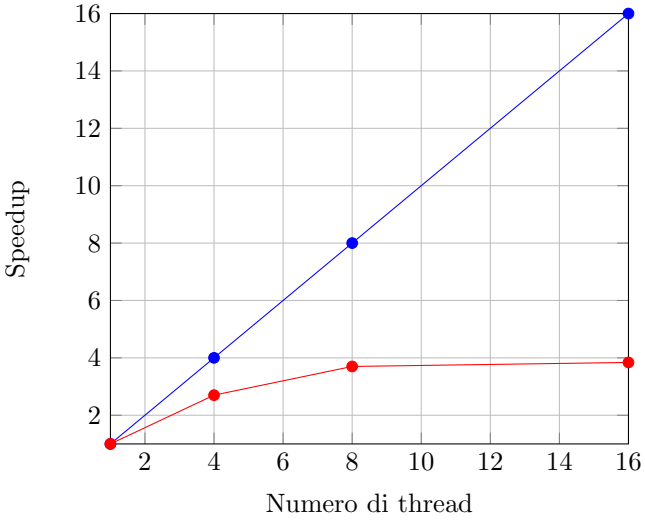
Per la taglia 1000000 abbiamo lo speed-up maggiore con 16 thread[5]:

$$Speedup = 0.0845/0.0220 \approx 384\%$$

$$Efficienza = (0.0845/0.0220)/16 \approx 24\%$$

[4] Valori presi da riga 16 e riga 17 dalla tabella MPI taglia 100000
[5] Valori presi da riga 1 e riga 15 dalla tabella MPI taglia 1000000

Speedup ideale (BLU) e speedup reale (ROSSO) taglia 1000000



Essendo la linearizzazione una operazione equivalente a una visita in order, ha una complessità computazionale pari ad $o(N)$ e quindi può beneficiare dei vantaggi della parallelizzazione. Inoltre notiamo dal grafico dello speedup, che la differenza tra 8 e 16 thread è praticamente nulla, il motivo è che la Cpu dove vengono eseguiti i test ha 6 thread (i test per 6 thread non sono potuti essere eseguiti, data la necessità che i thread siano potenza di due).

Di seguito vengono riportati i risultati per le ottimizzazioni O1,O2 ed O3.

5.3 Ottizzazione 1

5.3.1 Taglia 10000 MPI

	processes	omp$_t$*hreads*	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.020705	3.9e-05	0.016212	0.004169	0.000171
1	1	4	0.02103	3.9e-05	0.016324	0.003975	0.000298
2	1	8	0.021375	4e-05	0.016269	0.00407	0.000637
3	1	16	0.02167	4.1e-05	0.016264	0.004042	0.001039
4	2	1	0.021793	0.000303	0.016926	0.004171	0.000173
5	2	4	0.02232	0.00034	0.017066	0.004172	0.000387
6	2	8	0.022526	0.000318	0.01694	0.00418	0.000733
7	2	16	0.022014	0.000272	0.016619	0.004173	0.000812
8	4	1	0.02167	0.000499	0.016686	0.004171	0.000172
9	4	4	0.022031	0.000481	0.016656	0.004173	0.000282
10	4	8	0.022263	0.000555	0.016724	0.004168	0.000498
11	4	16	0.022958	0.000514	0.016867	0.004179	0.000621
12	6	1	0.028841	0.000698	0.022746	0.004183	0.000169
13	6	4	0.027874	0.000746	0.021731	0.004169	0.000279
14	6	8	0.028422	0.00073	0.020765	0.004179	0.000386
15	6	16	0.028931	0.000733	0.022166	0.004179	0.000611
16	8	1	0.031543	0.002132	0.019159	0.004177	0.00017
17	8	4	0.031359	0.002002	0.02022	0.004163	0.000274
18	8	8	0.031021	0.001912	0.020736	0.004186	0.000381
19	8	16	0.03226	0.002487	0.019894	0.004181	0.000612

5.3.2 Taglia 10000 CUDA

	threads	blocks	omp$_t$*hreads*	total$_t$*ime*	gpu$_t$*ime*	cuda search	tree$_c$*reation*	tree$_l$*inearizatio*
0	256	40	1	0.123565	0.000610	0.119543	0.004022	0.000065
1	256	40	4	0.125429	0.000521	0.121286	0.004082	0.000796
2	256	40	8	0.120021	0.000509	0.115851	0.004064	0.000660
3	256	40	16	0.123400	0.000517	0.119244	0.004111	0.000822
4	512	20	1	0.127220	0.000590	0.123073	0.004122	0.000171
5	512	20	4	0.123488	0.000509	0.119293	0.004098	0.000756
6	512	20	8	0.123136	0.000506	0.119045	0.004045	0.000864
7	512	20	16	0.124660	0.000507	0.120511	0.004064	0.000809
8	1024	10	1	0.124628	0.000586	0.120395	0.004163	0.000256
9	1024	10	4	0.123822	0.000521	0.119667	0.004111	0.000778
10	1024	10	8	0.122128	0.000508	0.118000	0.004043	0.000723
11	1024	10	16	0.122818	0.000513	0.118614	0.004136	0.000795

5.3.3 Taglia 10000 Seriale

	0
total_time	0.023
$\text{binary}_search_time$	0.019
$\text{tree}_creation_time$	0.003

5.3.4 Taglia 100000 MPI

	processes	$\text{omp}_threads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.076667	0.000275	0.02033	0.052122	0.002419
1	1	4	0.076112	0.00027	0.020252	0.052235	0.0018
2	1	8	0.076102	0.000263	0.020307	0.052147	0.001613
3	1	16	0.077023	0.000261	0.020234	0.052353	0.00238
4	2	1	0.080259	0.000789	0.023208	0.052244	0.002379
5	2	4	0.07988	0.000754	0.02355	0.052096	0.001718
6	2	8	0.079943	0.000708	0.0232	0.052078	0.001744
7	2	16	0.080431	0.000654	0.023164	0.052065	0.002396
8	4	1	0.082272	0.001346	0.024189	0.052201	0.002391
9	4	4	0.080871	0.001463	0.023416	0.052174	0.001485
10	4	8	0.081738	0.001285	0.023544	0.052367	0.001614
11	4	16	0.082091	0.001028	0.024145	0.052112	0.001925
12	6	1	0.087659	0.002314	0.027838	0.052337	0.002397
13	6	4	0.085989	0.00256	0.025855	0.052468	0.001666
14	6	8	0.085323	0.002676	0.025341	0.0531	0.001764
15	6	16	0.085667	0.002027	0.025955	0.052456	0.00244
16	8	1	0.095829	0.003812	0.031993	0.052469	0.002343
17	8	4	0.096657	0.003916	0.032725	0.05283	0.001687
18	8	8	0.09608	0.00252	0.032644	0.052081	0.001671
19	8	16	0.095405	0.002641	0.023613	0.05262	0.001532

5.3.5 Taglia 100000 CUDA

	threads	blocks	$\text{omp}_threads$	total_time	gpu_time	cuda search	$\text{tree}_creation$	$\text{tree}_linearization$
0	256	391	1	0.168	0.000571	0.112	0.054	0.002
1	256	391	4	0.169	0.000532	0.111	0.055	0.001
2	256	391	8	0.167	0.000538	0.109	0.055	0.001
3	256	391	16	0.164	0.000556	0.107	0.055	0.001
4	512	196	1	0.169	0.0006	0.114	0.054	0.002
5	512	196	4	0.167	0.00053	0.11	0.055	0.001
6	512	196	8	0.167	0.000534	0.109	0.055	0.001
7	512	196	16	0.166	0.000524	0.107	0.055	0.0
8	1024	98	1	0.17	0.00058	0.112	0.055	0.002
9	1024	98	4	0.168	0.000516	0.111	0.055	0.001
10	1024	98	8	0.167	0.000534	0.109	0.055	0.001
11	1024	98	16	0.165	0.00054	0.11	0.054	0.001

5.3.6 Taglia 100000 Seriale

	0
total_time	0.08
$\text{binary}_search_time$	0.024
$\text{tree}_creation_time$	0.055

5.3.7 Taglia 1000000 MPI

	processes	$\text{omp}_threads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	1.167605	0.001245	0.024607	1.024427	0.075676
1	1	4	1.121268	0.001243	0.024379	1.025084	0.028737
2	1	8	1.118743	0.001239	0.024502	1.028586	0.021385
3	1	16	1.109923	0.001241	0.024501	1.020945	0.021011
4	2	1	1.179687	0.002204	0.029849	1.028597	0.076326
5	2	4	1.131731	0.002201	0.030495	1.027955	0.02864
6	2	8	1.120599	0.00218	0.029206	1.024983	0.021266
7	2	16	1.129684	0.002145	0.030613	1.030604	0.021236
8	4	1	1.185191	0.003953	0.035508	1.025627	0.075866
9	4	4	1.139413	0.003774	0.033225	1.027414	0.028641
10	4	8	1.132286	0.003855	0.036465	1.028118	0.02132
11	4	16	1.135647	0.003945	0.032753	1.027409	0.020867
12	6	1	1.18912	0.005291	0.033459	1.030303	0.075585
13	6	4	1.140449	0.005353	0.03185	1.025941	0.028686
14	6	8	1.139493	0.004797	0.033512	1.035814	0.021322
15	6	16	1.127552	0.004672	0.033195	1.025155	0.020887
16	8	1	1.195703	0.006946	0.03509	1.020539	0.076088
17	8	4	1.151638	0.013592	0.036491	1.023594	0.028754
18	8	8	1.144722	0.012704	0.038238	1.026777	0.021374
19	8	16	1.145153	0.011649	0.031507	1.028505	0.020772

5.3.8 Taglia 1000000 CUDA

	threads	blocks	$\text{omp}_threads$	total_time	gpu_time	cuda search	$\text{tree}_creation$	$\text{tree}_linearization$
0	256	3907	1	1.314	0.001779	0.177	1.083	0.058
1	256	3907	4	1.283	0.001637	0.143	1.091	0.023
2	256	3907	8	1.279	0.001676	0.136	1.097	0.017
3	256	3907	16	1.268	0.001687	0.137	1.08	0.016
4	512	1954	1	1.309	0.001784	0.18	1.084	0.058
5	512	1954	4	1.276	0.001628	0.143	1.085	0.023
6	512	1954	8	1.277	0.001711	0.134	1.099	0.017
7	512	1954	16	1.272	0.001692	0.132	1.083	0.016
8	1024	977	1	1.314	0.001887	0.18	1.08	0.058
9	1024	977	4	1.275	0.001621	0.143	1.084	0.023
10	1024	977	8	1.271	0.001639	0.136	1.081	0.017
11	1024	977	16	1.265	0.001719	0.135	1.084	0.017

5.3.9 Taglia 1000000 Seriale

	0
total$_t ime$	1.148
binary$_s earch_t ime$	0.027
tree$_c reation_t ime$	1.12

5.4 Ottizzazione 2

5.4.1 Taglia 10000 MPI

	processes	omp$_t hreads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.020385	4e-05	0.016046	0.004026	0.000151
1	1	4	0.020668	4.1e-05	0.01608	0.003928	0.000313
2	1	8	0.020734	4e-05	0.016117	0.003924	0.000376
3	1	16	0.021284	4.1e-05	0.016046	0.003956	0.00078
4	2	1	0.021399	0.000263	0.016784	0.004004	0.000152
5	2	4	0.021624	0.000306	0.016569	0.004026	0.000307
6	2	8	0.021784	0.000224	0.016685	0.004019	0.000356
7	2	16	0.022398	0.000298	0.016587	0.004021	0.000972
8	4	1	0.022038	0.000514	0.017082	0.003994	0.000151
9	4	4	0.021791	0.000492	0.016327	0.004024	0.000269
10	4	8	0.021612	0.00047	0.016555	0.004013	0.000376
11	4	16	0.022422	0.000542	0.016769	0.004021	0.000636
12	6	1	0.028288	0.000755	0.021473	0.004028	0.000156
13	6	4	0.029055	0.000983	0.021795	0.004018	0.000272
14	6	8	0.027821	0.000749	0.021456	0.004028	0.000371
15	6	16	0.028727	0.000743	0.021559	0.004004	0.0006
16	8	1	0.030095	0.002058	0.023249	0.004006	0.000152
17	8	4	0.030767	0.00286	0.021997	0.004009	0.000267
18	8	8	0.030928	0.001903	0.022242	0.00403	0.000388
19	8	16	0.030659	0.002163	0.020438	0.004007	0.000601

5.4.2 Taglia 10000 CUDA

	threads	blocks	omp$_t hreads$	total$_t ime$	gpu$_t ime$	cuda search	tree$_c reation$	tree$_l inearizatio$
0	256	40	1	0.127333	0.000605	0.123250	0.004062	0.000104
1	256	40	4	0.126837	0.000534	0.122735	0.004061	0.000714
2	256	40	8	0.127109	0.000529	0.122978	0.004065	0.000761
3	256	40	16	0.131500	0.000545	0.127312	0.004167	0.000896
4	512	20	1	0.134667	0.000584	0.130429	0.004119	0.000310
5	512	20	4	0.131021	0.000545	0.126792	0.004125	0.000708
6	512	20	8	0.125111	0.000534	0.121022	0.004022	0.000822
7	512	20	16	0.126468	0.000522	0.122319	0.004064	0.000851
8	1024	10	1	0.127372	0.000576	0.123326	0.004023	0.000279
9	1024	10	4	0.131915	0.000530	0.127787	0.004085	0.000766
10	1024	10	8	0.129889	0.000564	0.125733	0.004111	0.000778
11	1024	10	16	0.128130	0.000531	0.123913	0.004152	0.000783

5.4.3 Taglia 10000 Seriale

	0
total$_t ime$	0.022
binary$_s earch_t ime$	0.018
tree$_c reation_t ime$	0.003

5.4.4 Taglia 100000 MPI

	processes	omp$_t hreads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.077008	0.00027	0.020061	0.053291	0.002036
1	1	4	0.075015	0.000271	0.020073	0.051877	0.001508
2	1	8	0.076244	0.000263	0.020079	0.052819	0.001638
3	1	16	0.076794	0.000267	0.020099	0.052545	0.001668
4	2	1	0.080717	0.000611	0.023437	0.052667	0.00207
5	2	4	0.080322	0.000743	0.023356	0.052575	0.001508
6	2	8	0.079632	0.000746	0.022974	0.052937	0.001578
7	2	16	0.080921	0.000607	0.022942	0.052428	0.002124
8	4	1	0.082085	0.001176	0.023474	0.052794	0.002069
9	4	4	0.080887	0.001287	0.023643	0.053161	0.001377
10	4	8	0.080547	0.001522	0.02373	0.052631	0.001484
11	4	16	0.082199	0.001837	0.023421	0.052623	0.002037
12	6	1	0.087955	0.001949	0.024533	0.053676	0.002074
13	6	4	0.086347	0.002141	0.02541	0.053308	0.001405
14	6	8	0.086318	0.002099	0.025666	0.053074	0.001551
15	6	16	0.086804	0.001991	0.024081	0.052487	0.00196
16	8	1	0.096597	0.003375	0.03094	0.053408	0.002001
17	8	4	0.094936	0.002631	0.029496	0.05265	0.001404
18	8	8	0.097851	0.005168	0.029739	0.053812	0.00142
19	8	16	0.097021	0.004009	0.032344	0.053325	0.001493

5.4.5 Taglia 100000 CUDA

	threads	blocks	omp$_t hreads$	total$_t ime$	gpu$_t ime$	cuda search	tree$_c reation$	tree$_l inearizatio$
0	256	391	1	0.169	0.000589	0.111	0.054	0.002
1	256	391	4	0.168	0.000505	0.111	0.055	0.001
2	256	391	8	0.167	0.000525	0.109	0.055	0.001
3	256	391	16	0.17	0.000588	0.113	0.054	0.001
4	512	196	1	0.172	0.000578	0.114	0.054	0.002
5	512	196	4	0.169	0.000508	0.112	0.055	0.001
6	512	196	8	0.165	0.000532	0.108	0.055	0.001
7	512	196	16	0.166	0.000559	0.109	0.055	0.001
8	1024	98	1	0.17	0.000599	0.112	0.055	0.002
9	1024	98	4	0.17	0.000508	0.113	0.055	0.001
10	1024	98	8	0.166	0.000564	0.108	0.055	0.001
11	1024	98	16	0.166	0.000539	0.109	0.055	0.001

5.4.6 Taglia 100000 Seriale

	0
total$_t ime$	0.082
binary$_s earch_t ime$	0.024
tree$_c reation_t ime$	0.057

5.4.7 Taglia 1000000 MPI

	processes	omp$_t hreads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	1.265083	0.00121	0.02435	1.146783	0.059493
1	1	4	1.227814	0.001224	0.024111	1.144378	0.023699
2	1	8	1.224115	0.00122	0.024212	1.145849	0.018454
3	1	16	1.224148	0.00124	0.024218	1.145808	0.017895
4	2	1	1.267168	0.002257	0.028495	1.143048	0.060141
5	2	4	1.232735	0.002161	0.028837	1.143384	0.023678
6	2	8	1.238149	0.002212	0.02764	1.153561	0.018437
7	2	16	1.226597	0.002221	0.029157	1.141571	0.017998
8	4	1	1.284133	0.00385	0.033784	1.149112	0.060169
9	4	4	1.240748	0.003734	0.03598	1.143819	0.023701
10	4	8	1.247939	0.00401	0.036541	1.150332	0.018033
11	4	16	1.238359	0.003829	0.035137	1.144349	0.017606
12	6	1	1.277792	0.005138	0.031086	1.143184	0.059652
13	6	4	1.241337	0.004402	0.029832	1.144149	0.0237
14	6	8	1.23312	0.004534	0.032252	1.139033	0.018396
15	6	16	1.236427	0.004777	0.032642	1.142094	0.018426
16	8	1	1.301096	0.009221	0.036013	1.152228	0.059852
17	8	4	1.258611	0.010369	0.03739	1.145154	0.023708
18	8	8	1.253057	0.010322	0.035759	1.147977	0.018294
19	8	16	1.256726	0.011597	0.033521	1.147752	0.018092

5.4.8 Taglia 1000000 CUDA

	threads	blocks	omp$_t hreads$	total$_t ime$	gpu$_t ime$	cuda search	tree$_c reation$	tree$_l inearizatio$
0	256	3907	1	1.326	0.001761	0.177	1.098	0.057
1	256	3907	4	1.281	0.00164	0.143	1.082	0.022
2	256	3907	8	1.274	0.001698	0.134	1.088	0.017
3	256	3907	16	1.269	0.001869	0.135	1.082	0.016
4	512	1954	1	1.316	0.001849	0.178	1.085	0.057
5	512	1954	4	1.279	0.001656	0.143	1.084	0.022
6	512	1954	8	1.271	0.001691	0.136	1.077	0.017
7	512	1954	16	1.277	0.001697	0.135	1.091	0.016
8	1024	977	1	1.306	0.001751	0.178	1.069	0.057
9	1024	977	4	1.275	0.001669	0.143	1.09	0.022
10	1024	977	8	1.283	0.001733	0.14	1.094	0.017
11	1024	977	16	1.27	0.001677	0.134	1.082	0.016

5.4.9 Taglia 1000000 Seriale

	0
total$_t ime$	1.293
binary$_s earch_t ime$	0.026
tree$_c reation_t ime$	1.266

5.5 Ottizzazione 3

5.5.1 Taglia 10000 MPI

	processes	omp$_t$*hreads*	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.020497	4e-05	0.016031	0.004167	0.000152
1	1	4	0.020778	3.9e-05	0.016092	0.004075	0.000375
2	1	8	0.020853	4.3e-05	0.016065	0.003961	0.00043
3	1	16	0.021619	4.2e-05	0.016014	0.004022	0.000843
4	2	1	0.021726	0.000356	0.016875	0.004155	0.000153
5	2	4	0.021972	0.000233	0.016998	0.004158	0.000429
6	2	8	0.021969	0.000263	0.016398	0.004151	0.000612
7	2	16	0.022104	0.000223	0.016732	0.004152	0.00062
8	4	1	0.0222	0.000537	0.016596	0.004148	0.000153
9	4	4	0.021897	0.00054	0.016186	0.004161	0.000272
10	4	8	0.02237	0.000502	0.016973	0.004147	0.000379
11	4	16	0.022852	0.000488	0.016821	0.004164	0.000621
12	6	1	0.028551	0.000846	0.022371	0.004169	0.000154
13	6	4	0.028556	0.00072	0.021556	0.004153	0.000264
14	6	8	0.028572	0.000739	0.021322	0.004159	0.00039
15	6	16	0.029424	0.000707	0.021276	0.004174	0.000598
16	8	1	0.032584	0.001982	0.020973	0.004159	0.000155
17	8	4	0.030546	0.001938	0.021612	0.004177	0.000272
18	8	8	0.030876	0.002042	0.020897	0.004166	0.000379
19	8	16	0.030989	0.002449	0.020329	0.004158	0.000602

5.5.2 Taglia 10000 CUDA

	threads	blocks	omp$_t$*hreads*	total$_t$*ime*	gpu$_t$*ime*	cuda search	tree$_c$*reation*	tree$_l$*inearizatio*
0	256	40	1	0.122489	0.000589	0.118400	0.004067	0.000067
1	256	40	4	0.123979	0.000538	0.119812	0.004146	0.000771
2	256	40	8	0.120571	0.000515	0.116449	0.004061	0.000673
3	256	40	16	0.121156	0.000513	0.117022	0.004089	0.000867
4	512	20	1	0.124422	0.000579	0.120222	0.004089	0.000089
5	512	20	4	0.122933	0.000519	0.118800	0.004044	0.000644
6	512	20	8	0.121458	0.000505	0.117375	0.004062	0.000792
7	512	20	16	0.122667	0.000493	0.118511	0.004089	0.000867
8	1024	10	1	0.124000	0.000560	0.119841	0.004136	0.000273
9	1024	10	4	0.123327	0.000539	0.119204	0.004122	0.000816
10	1024	10	8	0.121935	0.000510	0.117783	0.004065	0.000717
11	1024	10	16	0.121870	0.000518	0.117652	0.004152	0.000848

5.5.3 Taglia 10000 Seriale

	0
total_time	0.023
$\text{binary}_search_time$	0.019
$\text{tree}_creation_time$	0.003

5.5.4 Taglia 100000 MPI

	processes	$\text{omp}_threads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	0.075762	0.000279	0.020049	0.052059	0.002059
1	1	4	0.075025	0.000269	0.020099	0.051824	0.001568
2	1	8	0.075132	0.000272	0.020037	0.051905	0.001534
3	1	16	0.075821	0.000263	0.020042	0.052065	0.00233
4	2	1	0.080493	0.000708	0.023748	0.051915	0.002066
5	2	4	0.07943	0.000729	0.023305	0.05183	0.001503
6	2	8	0.07894	0.000757	0.022933	0.051784	0.001591
7	2	16	0.07998	0.000652	0.023661	0.051669	0.00215
8	4	1	0.083341	0.001295	0.024442	0.052144	0.002067
9	4	4	0.081502	0.001537	0.024476	0.051816	0.001423
10	4	8	0.080652	0.001419	0.023704	0.052049	0.001509
11	4	16	0.081366	0.001127	0.023544	0.051933	0.002231
12	6	1	0.0866	0.001976	0.026597	0.052521	0.002063
13	6	4	0.084759	0.002156	0.024341	0.051832	0.001528
14	6	8	0.08544	0.001988	0.026947	0.051886	0.001354
15	6	16	0.086798	0.001992	0.025024	0.052492	0.001922
16	8	1	0.095795	0.003781	0.029228	0.052442	0.002025
17	8	4	0.096105	0.003575	0.033418	0.052406	0.001432
18	8	8	0.095129	0.003618	0.031551	0.052695	0.001233
19	8	16	0.094058	0.003839	0.032812	0.052046	0.00198

5.5.5 Taglia 100000 CUDA

	threads	blocks	$\text{omp}_threads$	total_time	gpu_time	cuda search	$\text{tree}_creation$	$\text{tree}_linearization$
0	256	391	1	0.168	0.000567	0.11	0.055	0.002
1	256	391	4	0.168	0.000509	0.111	0.055	0.001
2	256	391	8	0.165	0.000534	0.108	0.055	0.001
3	256	391	16	0.167	0.000506	0.111	0.054	0.001
4	512	196	1	0.169	0.000592	0.112	0.054	0.002
5	512	196	4	0.17	0.000532	0.113	0.055	0.001
6	512	196	8	0.165	0.000513	0.109	0.055	0.001
7	512	196	16	0.167	0.000544	0.111	0.053	0.001
8	1024	98	1	0.171	0.000606	0.113	0.055	0.002
9	1024	98	4	0.171	0.000512	0.111	0.055	0.001
10	1024	98	8	0.164	0.000512	0.107	0.055	0.001
11	1024	98	16	0.167	0.000541	0.111	0.053	0.001

5.5.6 Taglia 100000 Seriale

	0
$\text{total}_t ime$	0.084
$\text{binary}_s earch_t ime$	0.025
$\text{tree}_c reation_t ime$	0.058

5.5.7 Taglia 1000000 MPI

	processes	$\text{omp}_t hreads$	total time	comm time	binary search	tree creation	tree linearization
0	1	1	1.141666	0.001232	0.024274	1.021282	0.060528
1	1	4	1.103102	0.001222	0.024127	1.019963	0.023708
2	1	8	1.099386	0.001235	0.024266	1.02187	0.01828
3	1	16	1.097563	0.001244	0.0243	1.019184	0.017723
4	2	1	1.142843	0.002285	0.029954	1.014796	0.060217
5	2	4	1.111575	0.002192	0.027993	1.019559	0.02359
6	2	8	1.103809	0.00216	0.0287	1.01672	0.018311
7	2	16	1.096407	0.002222	0.030282	1.012145	0.01826
8	4	1	1.160827	0.003762	0.031778	1.024672	0.060138
9	4	4	1.116744	0.003577	0.03071	1.022435	0.023858
10	4	8	1.122271	0.003863	0.035685	1.027149	0.01836
11	4	16	1.122317	0.003842	0.03663	1.026521	0.018174
12	6	1	1.160691	0.004595	0.033887	1.024001	0.059807
13	6	4	1.115811	0.004766	0.029231	1.020136	0.023773
14	6	8	1.116723	0.004727	0.030494	1.02276	0.018421
15	6	16	1.11617	0.005144	0.032496	1.024437	0.018014
16	8	1	1.171149	0.010764	0.034438	1.022634	0.059877
17	8	4	1.13525	0.010103	0.035413	1.023346	0.023745
18	8	8	1.127307	0.010199	0.038721	1.01914	0.018353
19	8	16	1.130348	0.010907	0.036975	1.025829	0.01797

5.5.8 Taglia 1000000 CUDA

	threads	blocks	$\text{omp}_t hreads$	$\text{total}_t ime$	$\text{gpu}_t ime$	cuda search	$\text{tree}_c reation$	$\text{tree}_l inearizatio$
0	256	3907	1	1.314	0.00177	0.18	1.082	0.057
1	256	3907	4	1.269	0.001639	0.142	1.08	0.022
2	256	3907	8	1.261	0.001687	0.136	1.078	0.017
3	256	3907	16	1.258	0.001788	0.135	1.079	0.017
4	512	1954	1	1.332	0.001828	0.18	1.088	0.057
5	512	1954	4	1.262	0.001626	0.144	1.077	0.022
6	512	1954	8	1.264	0.001671	0.138	1.079	0.017
7	512	1954	16	1.266	0.001654	0.135	1.08	0.016
8	1024	977	1	1.3	0.001828	0.181	1.076	0.058
9	1024	977	4	1.269	0.001636	0.144	1.078	0.022
10	1024	977	8	1.265	0.001695	0.138	1.078	0.017
11	1024	977	16	1.262	0.001695	0.134	1.077	0.016

5.5.9 Taglia 1000000 Seriale

	0
total_time	1.184
$\text{binary}_search_time$	0.028
$\text{tree}_creation_time$	1.156

5.6 Conclusioni

Per le diverse ottimizzazioni non ci sono cambi sostanziali tra i rapporti di velocità[6]. Considerando le approssimazioni e il numero di test eseguiti, possiamo concludere che per le tre taglie prese in considerazione la versione seriale dell'algoritmo performa sempre meglio, tenendo anche in considerazione che prima di eseguire la ricerca binaria su MPI linearizziamo l'albero, e che quindi l'algoritmo di ricerca sarà più veloce dato che impiegerà esattamente $log(n)$ step, e non $o(logn)$ come nella versione seriale. Tra le versione parallele, per le prime due taglie, MPI risulta notevolmente più veloce di CUDA, mentre per la terza taglia CUDA e MPI sono simili. Questo è spiegabile dal fatto che la funzione di ricerca per CUDA (il campo cuda search) ha una durata che è quasi completamente indipendente dalla taglia dell'input (ma rimane la dipendenza data dalla linerizzazione e il trasferminto dell'albero dall'host al device). Questo potrebbe suggerire che la versione CUDA potrebbe ottenere risultati migliori della versione seriale, ma eseguendo alcune iterazione dell'algoritmo, questo non sembra risultare vero neanche per un albero di 1Gb[7] (risultato aspettabile data la nutara logaritmica del problema). CUDA può ottenere dei risultati migliori quando l'operazione di comprazione tra dati è molto complessa (visto che cuda controlla con una memcmp), ma proseguire per questa strada sarebbe una forzatura, perché qualsiasi sia il dato gli si può sempre associare una chiave univoca intera da usare per le comporazioni.

[6] Per alcuni campi sembra che MPI sia leggermente meglio del seriale, ma il motivo è dovuto solo al tempo di creazione dell'albero

[7] risultati disponibili nella cartella TEST1GB, risultati di 5 iterazioni ottimizzazione O3

6 Come eseguire il codice

Il codice per essere compilato ha bisogno di una libreria MPI, di OpenMP, del compilatore Nvidia NVCC, di una GPU Nvidia e di Python. Se si intende anche generare i CSV da cui sono state create le cartelle c'è anche bisogno della libreria Python Panda (installabile con il comando 'pip install pandas'). Inoltre prima di provare ad eseguire il codice è necessario modificare le variabili nel Makefile per assicurarsi che sono compatibili col sistema dove verrà compilato e eseguito il progetto. Ad esempio, io utilizzo un sistema Windows, ed utilizzo come libreria MPI la libreria MPI di Microsoft, e quindi le variabili per il mio sistema sono definite in questo modo:

```
# COMPILATORE C
CC = C:/MinGW64/mingw64/bin/gcc.exe
# MPI WRAPPER (SE PRESENTE ASSICURARSI CHE IL WRAPPER UTILIZZI LO STESSO COMPILATORE DI CC)
MPICC = C:/MinGW64/mingw64/bin/gcc.exe
# FLAG MPI (SE NON SI USA IL WRAPPER CONFIGURARLE CON I PATH CORRETTI, ALTRIMENTI LASCIARE VUOTA)
MPICC_FLAGS = -I"C:\\Program Files (x86)\\Microsoft SDKs\\MPI\\Include\\" -L"C:\\Program Files (x86)\\Microsoft SDKs\\MPI\\Lib\\x64\\" -lmsmpi
# IL COMANDO PER ESEGUIRE MPI
EXECUTE = mpiexec -n
NVCC = nvcc
# LA MIA GPU HA UNA CC 8.6 QUINDI:
NVCC_FLAGS = -arch=compute_86 -code=sm_86
```

Inoltre si possono anche modificare le variabile per l'esecuzione dei test, cambiando il numero di nodi e il numero di iterazioni. I test sono stati eseguiti con le seguente variabili:

```
NODES = 10000 100000 1000000
PROCESSES = 1 2 4 6 8
# LA VARIABILE OMP_THREADS DEVE ESSERE UNA POTENZA DI DUE
OMP_THREADS = 1 4 8 16
ITERATION = 50
# IL CODICE E' STATO TESTATO PER QUESTE OTTIMIZZAZIONI
# SI POSSONO TESTARE ANCHE ALTRE OTTIMIZZAZIONI, MA NON GARANTISCO IL FUNZIONAMENTO
OPTIMIZATIONS = O0 O1 O2 O3
# PER CUDA
THREADS_PER_BLOCK = 256 512 1024
```

Quindi, dopo aver modificato le variabile del Makefile, per eseguire il codice:

1. Locarsi nella cartella del progetto dove è presente il file Makefile
2. Eseguire da terminale il comando 'make all', per compilare e generare gli eseguibili dell'intero progetto
3. Eseguire da terminale il comando 'make test', per eseguire i test

4. A questo punto la cartella 'results' dovrebbe essere popolata con i CSV con i dati grezzi, divisi per taglia ottimizzazione e tipo.

5. A questo punto la cartella 'tables' dovrebbe essere popolata con i CSV con i dati aggregati, divisi per taglia ottimizzazione e tipo.

6. Eseguire da terminale il comando 'make clean', se si intende cancellare la build del progetto, i risultati grezzi e le tabelle.

7 Codice Sorgente

La documentazione del codice sorgente ed il codice sorgente stesso è disponibile anche online al seguente link: https://hpcunisa20232024sessa.surge.sh/.

Di seguito riporto le parti di codice che considero più importanti.

7.1 OpenMP, Linearizzazione

```
1     /**
2  * @brief Creates a sorted array of the nodes in the tree and
       fills the mydata
3  * array passed as paramaters with the nodes data. Uses OpenMP
       if threads > 1
4  * @param rbt Pointer to the Red-black Tree structure
5  * @param threads Number of threads to use in the linearization
       process.
6  * Has to be a power 2.
7  * @param data_array Pointer to the mydata array that has to be
       filled with the
8  * node's data.
9  * @return Retruns a sorted array of the nodes contained in the
       tree
10 * @note before running rb_node_array set_counts should be
       runned
11 */
12 rbnode **rb_node_array(rbtree *rbt, int threads, mydata *
       data_array)
13 {
14
15     /* index to traverse the tree */
16     int index = 0;
17     /* sorted array of the node of the tree to be returned */
18     rbnode **result_array = (rbnode **)malloc(sizeof(rbnode *)
           * rbt->count + 1);
19
20     if (threads == 1)
21     {
22         traversal(rbt, RB_FIRST(rbt), result_array, data_array,
               &index);
```

```
23          return result_array;
24      }
25
26      /* num of nodes that are above the level log2(threads) */
27      int excluded_nodes_num = threads - 1;
28
29      rbnode **excluded_nodes = (rbnode **)malloc(sizeof(rbnode
            *) * excluded_nodes_num);
30      rbnode **included_roots = (rbnode **)malloc(sizeof(rbnode
            *) * threads);
31
32      /* per assumption this will be a power two so log2(threads)
             will be an integer */
33      int level_searched = log2(threads);
34      int included_index = 0;
35      int excluded_index = 0;
36
37      /* set the subroots from where to start the parallel
            execution of trasversal + save the excluded nodes*/
38      set_roots(rbt, RB_FIRST(rbt), level_searched, 0,
            included_roots, excluded_nodes, &included_index, &
            excluded_index);
39      /* sort the excluded nodes */
40      sort_node_array(rbt, excluded_nodes, excluded_nodes_num);
41
42
43      /* offset at which each thread should start writing on the
            result_array*/
44      int *offset = (int *)malloc(sizeof(int) * threads);
45      offset[0] = 0; /* first thread starts from 0*/
46
47      /* putting the excluded nodes in the right place in the
            result_array and data_array + setting the offset*/
48      for (int i = 1; i < threads; i++)
49      {
50          /* the offset of the i-thread is the offset of the i-1
                thread + the number of nodes it will handle plus one
51          because between each each area that will be filled by
                the thread there is an excluded node to be put in*/
52          offset[i] = offset[i - 1] + included_roots[i - 1]->
                count + 1;
53          // putting in the excluded node
54          result_array[offset[i - 1] + included_roots[i - 1]->
                count] = excluded_nodes[i - 1];
55          // putting in the data of the excluded nodes
56          data_array[offset[i - 1] + included_roots[i - 1]->count
                ] = *(mydata *)excluded_nodes[i - 1]->data;
57      }
58
59      /* Using omp parallel to share the work between threads,
```

```
            each threads start from a different node  based on his
            id*/
60  #pragma omp parallel shared(result_array, offset,
        included_roots) private(index) num_threads(threads)
61      {
62          /* index is private, so each thread can modify it*/
63          /* result_array can be shared because threads will not
                overlap each others, same thing for offset and
                included_roots, who are
64          also read only */
65
66          // each thread gets its id
67          int thread_id = omp_get_thread_num();
68
69          // each thread gets its node from where to start
70          rbnode *myroot = included_roots[thread_id];
71
72          // each thread sets the right value from where to start
                 for the index, so to not overlap eachothers
73          index = offset[thread_id];
74
75          // call to the in-order traversal, that also fills the
                result and data array
76          traversal(rbt, myroot, result_array, data_array, &index
                );
77      }
78
79      return result_array;
80  };
```

```
1      /**
2   * @brief Helper function for rb_node_array. It fills two array
        , one with the nodes at a certain level and the other with
        the nodes at a previous level
3   * @param rbt Pointer to the Red-black Tree structure
4   * @param node Pointer to the Root of the Red-black tree
5   * @param level Level where to take the nodes to fill the
        node_array
6   * @param currentLevel level from where to start the operation
        (It has to be 0)
7   * @param node_array array to contain the pointer of the nodes
        at the searched level
8   * @param excluded array to contain the pointer of the nodes at
         the previous levels
9   * @param included_index pointer to the included_index. (It has
         to start with an a value of 0)
10  * @param exclude_index pointer to the exclude_index. (It has
        to start with an a value of 0)
11  *
12  */
13  void set_roots(rbtree *rbt, rbnode *node, int level, int
        currentLevel, rbnode **node_array, rbnode **excluded, int *
        included_index, int *excluded_index)
14  {
15      // if the node is null or we surpassed the current level we
            are done
16      // base case
17      if (node == RB_NIL(rbt) || currentLevel > level)
18          return;
19
20      if (currentLevel == level)
21      {
22          // first the node_array is accessed at the value
                included_index points to and then gets increased
23          node_array[(*included_index)++] = node;
24          return;
25      }
26      // we also save the excluded roots
27      excluded[(*excluded_index)++] = node;
28      // recursive call, we go down a level, so currentLevel + 1
29      set_roots(rbt, node->left, level, currentLevel + 1,
            node_array, excluded, included_index, excluded_index);
30      set_roots(rbt, node->right, level, currentLevel + 1,
            node_array, excluded, included_index, excluded_index);
31  }
```

7.2 CUDA

```
1     /**
2  * @brief Cuda version of rb_find
3  * @param rbt pointer to the red-black-tree
4  * @param data pointer to the searched data
5  * @return Returns pointer to the node that contains the
       searched data, NULL if not found
6  */
7  rbnode *d_rb_find(rbtree *rbt, void *data)
8  {
9    /* time variables */
10   cudaEvent_t start, stop;
11   clock_t start_d, end_d;
12
13   /* we set the count variable of the nodes at log2(
         omp_num_threads),
14     the rbt tree implementation could be changed so that this
         is an operation done during the insert, but
15     this was not done, so I have to call this function. I
         choose to not time it in the tree linearization */
16   set_counts(rbt,RB_FIRST(rbt),log2(omp_threads),0);
17   start_d = clock();
18   cuda_search_start = clock();
19   //mydata *h_data_array = (mydata*)malloc(sizeof(mydata) * rbt
         ->count );
20   mydata *h_data_array;
21   cudaMallocHost((void **)&h_data_array, ( sizeof(mydata) * rbt
         ->count ));
22   cudaCheckError();
23   //linearizing tree
24   rbnode **h_node_array = rb_node_array(rbt, omp_threads,
         h_data_array);
25
26   end_d = clock();
27   tree_linearization_time =  ((double)(end_d - start_d)) /
         CLOCKS_PER_SEC;
28
29   /*
30    * the node h_node_array[i] contains data that has the same
        value as
31    * the data h_data_array[i]
32    * mydata * test =(mydata*) h_node_array[3]->data;
33    * printf("h_node_array[3] contiene : %d\n",test->key);
34    * printf("h_data_array[3] contiene : %d\n",h_data_array[3].
        key);
35    */
36
37   /* device variables */
38   mydata *d_data_array;
```

```
39
40    /* start timing the search */
41    cudaEventCreate(&start);
42    cudaEventCreate(&stop);
43    cudaEventRecord(start);
44
45    /* device allocations */
46    cudaMalloc((void **)&d_data_array, rbt->count * sizeof(mydata
          ));
47    cudaCheckError();
48
49    /* copying the data from host to device */
50    cudaMemcpy(d_data_array, h_data_array, rbt->count * sizeof(
          mydata), cudaMemcpyHostToDevice);
51    cudaCheckError();
52
53    /* setting up the kernel */
54    /* number of threads for each block, we try 256,512,1024*/
55    int threadsPerBlock = number_of_threads_per_block;
56    /* we are working with a one dimensional array, so the blocks
           will be one dimension also */
57    int blocksPerGrid = (rbt->count + threadsPerBlock - 1) /
          threadsPerBlock;
58    /* to write the result to file later*/
59    number_of_blocks = blocksPerGrid;
60
61    /* invoking kernel */
62    /* the device where to code has been tested has 8GB of memory
           and enough thread to handle every input test size,
63    so only one execution of the kernel is necessary */
64    d_find<<<blocksPerGrid, threadsPerBlock>>>(d_data_array, (*(
          mydata *)data));
65    cudaDeviceSynchronize();
66    cudaCheckError();
67
68    /* copying the result on the host result array*/
69    int found_index;
70    rbnode *ret = NULL;
71    /* copying from the device symbol (like a global variable for
           the device)*/
72    cudaMemcpyFromSymbol(&found_index, d_found_index, sizeof(int)
          );
73    cudaCheckError();
74
75    if (found_index != -1)
76      ret = h_node_array[found_index];
77
78    /* ret points to the searched node (if found), at this point
          d_rb_find is done the rest is just timing and frees*/
79    /* end timing */
```

```
80    cudaEventRecord(stop);
81    cudaEventSynchronize(stop);
82
83    float gpu_time;
84    cudaEventElapsedTime(&gpu_time, start, stop);
85    gpu_time_sec = gpu_time / 1000.0;
86
87    cudaEventDestroy(start);
88    cudaEventDestroy(stop);
89    cudaFree(d_data_array);
90    cudaFree(h_data_array);
91    free(h_node_array);
92
93    return ret;
94  }
95
96  /**
97   * @brief Kernel, Every thread checks a node doing a memcmp,
        bypassing the normal compare function
98   * @param data_array array of the data contained in the red-
        black tree.
99   * @param searched_data the searched data.
100  * @note Every thread checks an element of the data array based
       on his index with memcmp, if it correspond to the searched
       data
101  * it changes the value of the device symbol d_found_index to
       its index, no need to worry about race conditions because
       the element is unique in the tree
102  */
103 __global__ void d_find(const mydata *data_array, const mydata
      searched_data)
104 {
105   int index = blockIdx.x * blockDim.x + threadIdx.x;
106
107
108   if (d_memcmp(&searched_data, &data_array[index], sizeof(
        mydata)) == SUCCESS)
109   {
110     d_found_index = index; //no need for atomExch, each element
          is unique
111   }
112
113   // this is faster, but less generic
114   // if(data_array[index].key == searched_data.key)
        d_found_index = index;
115 }
116
117 /**
118  * @brief memcmp implementation for the device
119  * @param s2 Pointer to the first memory block to be compared
```

```
120  * @param s1 Pointer to the second memory block to be compared
121  * @param n Number of bytes to be compared.
122  * @note memcmp is public domain
123  */
124 __device__ int d_memcmp(const void *s1, const void *s2, size_t
       n)
125 {
126   const unsigned char *us1 = (const unsigned char *)s1;
127   const unsigned char *us2 = (const unsigned char *)s2;
128   while (n-- != 0)
129   {
130     if (*us1 != *us2)
131     {
132       return -1; // we do not care about order so i avoided the
             if that was here
133     }
134     us1++;
135     us2++;
136   }
137   return SUCCESS;
138 }
```

7.3 MPI

```
1  mydata *mypart = (mydata *)malloc(sizeof(mydata) *
       elem_per_process + 1);
2
3    if (rank == 0)
4      start_time = MPI_Wtime();
5    /* this are collective operations, so each process has to
        call this functions */
6    /* broadcast the searched data */
7    MPI_Bcast(&query, 1, MPI_MYDATA, 0, MPI_COMM_WORLD);
8    /* scatter the data_array, each process receives only part of
         the data_array */
9    MPI_Scatterv(data_array, count, displacementsv, MPI_MYDATA,
        mypart, count[rank], MPI_MYDATA, 0, MPI_COMM_WORLD);
10
11   if (rank == 0)
12   {
13     end_time = MPI_Wtime();
14     communication_time += end_time - start_time;
15     free(data_array);
16   }
17
18   start_time = MPI_Wtime();
19   // the binary search returns the index where the searched
        data was found
20   int index = binarySearch(mypart, query, 0, count[rank]);
```

```
21    end_time = MPI_Wtime();
22    binary_search_time = end_time - start_time;
23
24    // rank0 is the master process, so it will check wich process
          has found the data
25    // in our case is always the last one, so that we can time
          the binary search for the worst case where the data is not
          found
26    // (worst case does not mean a lot in MPI, because each
          process has to wait for the slowest process to finish to
          terminate the execution)
27    if (rank == 0)
28    {
29      start_time = MPI_Wtime();
30      MPI_Gather(&index, 1, MPI_INT, result_array, 1, MPI_INT, 0,
            MPI_COMM_WORLD);
31    }
32    else
33    {
34      MPI_Gather(&index, 1, MPI_INT, NULL, 0, MPI_INT, 0,
            MPI_COMM_WORLD);
35      // the processes that are not rank0 can terminate now
36      free(mypart);
37      MPI_Finalize();
38      return 0;
39    }
40    //int founded_flag = -1;
41    if (rank == 0)
42    {
43      end_time = MPI_Wtime();
44      communication_time += end_time - start_time;
45      for (int i = 0; i < number_of_processes; i++)
46      {
47        // iterate over the result array, if a process has not
              found the element it sends -1
48        // when the element is different than -1 it means the
              element was found
49        if (result_array[i] != -1)
50        {
51          //printf("FOUND BY %d",i);
52          // Accounting for displacements
53          index = result_array[i] + displacementsv[i];
54
55          end_time = MPI_Wtime();
56
57          /* verify the result with sequential implementation */
58          total_time = end_time - total_start;
59          rbnode *to_assert = rb_find(rbt, &query);
60          assert(to_assert == node_array[index]);
61          //founded_flag = 1;
```

```
62            break;
63          }
64        }
65    }
```

7.4 Crediti

Crediti per la macro cudaCheckError(): https://gist.github.com/jefflarkin/5390993[2]

Crediti per l'implementazione del Red-Black Tree in C: https://github.com/xieqing/red-black-tree[3]

References

[1] dvasavda. *Parallel Binary Search with OpenMP*. `https://github.com/dvasavda/openmp-binary-search/blob/master/Report.pdf`.

[2] jefflarkin. *cudaCheckError.c*. `https://gist.github.com/jefflarkin/5390993`.

[3] xieqing. *A Red-black Tree Implementation In C*. `https://github.com/xieqing/red-black-tree`.