ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

CAMPUS DI CESENA
SCUOLA DI SCIENZE

Corso di Laurea in Ingegneria e Scienze Informatiche

IMPLEMENTAZIONE CUDA SU GPU DI UN ALGORITMO SORT-BASED PER LA DISTRIBUZIONE EFFICIENTE DI DATI IN SIMULAZIONI DISTRIBUITE

Elaborata nel corso di: Sicurezza delle Reti

Tesi di Laurea di:
GIOVANNI POGGI

Relatore:
GABRIELE D'ANGELO

Co-relatore:
MORENO MARZOLLA

ANNO ACCADEMICO 2021–2022
SESSIONE III

PAROLE CHIAVE

Data Distribution Management
High Level Architectur
Interest Matching
Parallel Algorithm
CUDA Architecture

Dedicato alle persone che mi hanno aiutano ad essere sempre me stesso.

Indice

Elenco delle figure

Capitolo 1

Introduzione

L'obiettivo di questa tesi è quello di elaborare, descrivere ed analizzare un algoritmo che risolva un problema di sort-matching all'interno di una simulazione, utilizzando una GPU su architettura CUDA.
Siccome gli argomenti trattati di seguito sono alquanto tecnici, è bene introdurre alcuni concetti fondamentali prima di proseguire, partendo dal concetto di simulazione.
Nel campo dell'informatica, il termine «simulazione» indica la riproduzione di un sistema o parte di esso in un ambiente controllato. Per sistema, in tal senso, si può intendere una varietà di concetti e applicazioni facilmente riconoscibili nel quotidiano, ad esempio nei videogiochi, dove si «simulano» delle azioni: il comportamento dei personaggi, i dialoghi e le loro scelte, e poi il volo, l'esplorazione, il combattimento, la gestione di un esercito o di una battaglia campale, ma anche l'amministrazione di un villaggio, una metropoli o di un'abitazione, di un gruppo familiare o l'evoluzione di un ecosistema o di un intero pianeta.
Chiaramente, le simulazioni così intese, hanno un campo di applicazione virtualmente illimitato: dalle operazioni di decollo e atterraggio per aerei o razzi spaziali alle prestazioni su strada delle auto da corsa, passando per le previsioni meteorologiche, oceanografiche ed evoluzionistiche, fino ai modelli comportamentali fra esseri umani e animali, e alle raffinate interazioni biochimiche fra piante, funghi, batteri, virus e altri microorganismi.
Tutte queste applicazioni, fondamentalmente, vengono svolte attraverso uno o più calcolatori. L'affidabilità della simulazione è strettamente collegata al livello di dettaglio del modello; maggiore è il dettaglio, chiaramente, mag-

giore sarà la potenza di calcolo richiesta. Se si prendono in considerazione le previsioni climatiche, ad esempio, per ottenere una simulazione affidabile sarà necessario impiegare i supercomputer più sofisticati al mondo, supportati da altre unità di calcolo: una simulazione complessa come quella climatica viene dunque suddivisa in varie simulazioni più semplici. Al contrario, ad esempio nelle simulazioni di sistemi socio-economici, dove più componenti separati vanno considerati insieme, diverse simulazioni di sottoinsiemi relativamente semplici ed omogenei vanno compresi entro una simulazione più complessa.
Il problema che ne emerge dunque è quello dell'interoperabilità: le piattaforme di calcolo da accoppiare e le simulazioni da queste operate, infatti, sono spesso fra loro eterogenee. Interoperabilità che viene risolta attraverso l'applicazione di uno standard per architetture di sistemi atti a gestire delle simulazioni distribuite: l'«High-Level Architecture» («HLA»), che permette a più simulazioni operate su differenti piattaforme di interagire efficacemente. In buona sintesi, l'HLA definisce le simulazioni che devono interagire con l'infrastruttura, le regole che devono rispettare e le informazioni da comunicare.
L'implementazione della HLA viene detta «Run-Time Infrastructure» («RTA»), che è il middleware operante fra i vari software delle piattaforme coinvolte. Al suo interno, il servizio di «Data Distribution Management» («DDM») si occupa di distribuire i dati in modo da ridurre quelli da processare in ogni singola piattaforma e il traffico della rete.
Svolta questa breve introduzione nel presente primo capitolo, nel secondo capitolo si svolgerà un'introduzione rispetto all'argomento principale, con cenni al «Data Distribution Management» («DDM»). Si introdurrà l'«High Level Architecture», di cui si da una definizione e quindi una descrizione di massima, dando particolare attenzione alla sua struttura.
Si prosegue con l'analisi dettagliata del servizio di DDM, la sua funzione e i principi chiave a cui deve attenersi, nonchè i suoi componenti denominati «federati»).
Nel terzo capitolo si effettua l'introduzione al «matching», spiegando nel dettaglio di cosa si occupa questa operazione e le differenze sostanziali tra «extent», «region», «update» e «subscription». Dopo un breve cenno al «routing space» si presentano alcuni degli algoritmi più diffusi ed impiegati nello svolgimento operativo dei servizi di DDM. Ponendo forte accento sugli algoritmi «Brute-Force Parallel», «Grid-Based Parallel» e l'algoritmo

preso in esame, il «Sort-Based Parallel», aggiungendo qualche cenno sulla parallelizzazione con CUDA.
Il quarto capitolo è dedicato all'ottimizzazione dell'algoritmo Sort-Based utilizzando l'architettura CUDA. Dopo una breve introduzione alla GPU e CUDA, si presentano tutte le modifiche sostanziali effettuate al Sort-Based ponendo particolare attenzione alla comprensione del codice.
Si prosegue con l'adattamento realizzato dell'algoritmo su GPU ed i dettagli implementativi, ponendo un'ulteriore attenzione ai problemi riscontrati durante la scrittura del codice.
Nel quinto capitolo si analizzano le prestazioni ottenute durante l'esecuzione dell'algoritmo utilizzato, paragonandolo ai tempi di esecuzione degli algoritmi precedentemente introdotti. Si descrivono i criteri di valutazione che hanno permesso di comprendere l'ottenimento del «goal» iniziale (ovvero il superamento o eguagliamento delle prestazioni del Sort-Based su CPU) e si presentano i risultati ottenuti, descrivendo le rappresentazioni grafiche corredate dalle relative tabelle numeriche per una più corretta comprensione. Si conclude con una serie di commenti finali.
Seguono le conclusioni con una riflessione e discussione dei risultati, corredati dai possibili sviluppi futuri dell'applicazione in oggetto alla tesi che permettono di raggiungere maggiori prestazioni ed efficienza.

Capitolo 2

Introduzione al Data Distribution Management

Questo capitolo introduce il Data Distribution Management (DDM) e la sua importanza strumentale nelle simulazioni in ambienti distribuiti.
Nel primo paragrafo si svolge una breve descrizione della High Level Architecture (HLA), di cui il DDM fa parte. Nel secondo paragrafo, dunque, si opera un'introduzione al DDM che possa permettere di affrontare la discussione del capitolo terzo con sufficiente cognizione degli elementi fondamentali che lo costituiscono.

2.1 Introduzione all'High Level Architecture

La HLA consente l'interoperabilità tra sistemi di simulazione eterogenei ed è uno standard di simulazione negli standard di simulazione per l'esecuzione in ambienti distribuiti (detti in gergo tecnico «distribuiti») [1].
Oggi giorno, molte delle simulazioni operate in vari campi di applicazione risultano tanto complesse, ampie e dettagliate da non poter essere eseguite da un singolo computer, ossia da un singolo processore («CPU core»). Di conseguenza, è necessario collegare tra loro diversi computer formando una rete di calcolatori (comunemente chiamato Cluster) che possono scambiare informazioni ed aggiornamenti di stato tra loro. Questo, a sua volta, permette l'aggregazione di risorse computazionali che determinano, se opportunamente sfruttate dal software di simulazione, la sua scalabilità. L'HLA, in breve, identifica il sistema di connessioni architetturali che per-

mette la creazione e gestione di questi ambienti di simulazione dove svariati computer, dotati di software tra loro compatibili, possono comunicare fra loro grazie ad una serie di regole e protocolli condivisi.

2.1.1 Struttura dell'HLA

Detto e considerato ciò, si noti che l'HLA è composta fondamentalmente da tre componenti (Vedi Figura 2.1):

- **«Interfaccia»**: Che definisce il modo in cui gli ambienti di simulazione conformi ad HLA interagiscono tramite un'infrastruttura run-time (RTI). Ricordiamo che l'RTI è un software di supporto che fornisce sei gruppi di servizi (la cui implementazione completa, peraltro, non è obbligatoria ma discrezionale, a seconda dei fini che si propone l'HLA) forniti come librerie programmabili ed interfacce (Ovvero API) come definito dalle specifiche contingenti della HLA.
- **«Object Model Template» (OMT)**: Indica quali informazioni vengono documentate e come sono scambiate tra le simulazioni.
- **«Rule set»**: Che regola lo svolgimento delle simulazioni in ogni singola fase; il DDM applicato in ogni singolo software dovrà conformarsi al set di regole impostato per l'HLA che raggruppa il cluster di hardware operanti.

In linguaggio tecnico, gli ambienti di simulazione vengono denominati «federati» ed a sua volta una raccolta di federati - ossia un insieme di ambienti di simulazione - viene definita «federazione».
Ogni simulazione, ogni federato, dunque, partecipa all'esecuzione di una federazione di simulazioni, ossia a una simulazione più ampia, che allo stesso tempo mantiene un livello di dettaglio che ne assicura l'affidabilità in termini di sistema complesso proprio grazie all'implementazione dell'OMT, che fornisce un modello comune in cui un set di regole viene applicato uniformemente.
A sua volta, l'RTI fornisce i servizi definiti nell'interfaccia di specificazione - «Interface Specification», o «IS» - a seconda dei dati ritenuti rilevanti ai fini della simulazione. Più in particolare [2]:

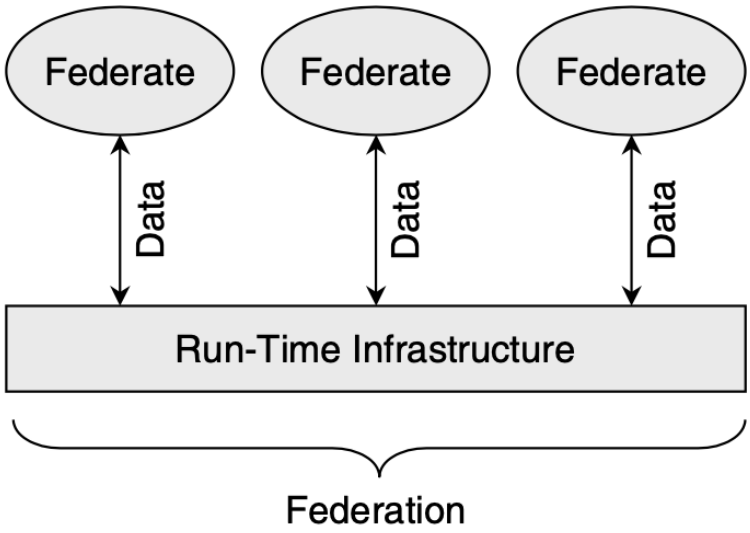

Figura 2.1: Rappresentazione concettuale di una federazione in HLA, dove i federati scambiano dati via RTI. [2]

«An HLA Run-Time Infrastructure (RTI) is a software implementation of the Interface Specification. The RTI actually provides the services defined in the Interface Specification, including services to start and stop a federation execution, to send data between interoperating federates, to control the amount and routing of data that is passed, and to coordinate the passage of simulated time among the federates».

Quindi, i federati - ovvero le singole simulazioni - svolgono queste funzioni invocando il servizio RTI appropriato a seconda degli obiettivi che si intendono raggiungere attraverso l'esecuzione della federazione di simulazioni, a loro volta definiti dalle indicazioni fornite nell'interfaccia di specificazione.
Fatta questa brevissima premessa, si può proseguire con la definizione di DDM e quali sono i suoi obiettivi principali.

2.2 Il Data Distribution Management

In una simulazione, almeno teoricamente, si può ottenere il pieno accesso alla memoria del software a cui è stato assegnato il compito di processare le informazioni. Di conseguenza, queste informazioni, ossia i dati oggetto di processamento, sono completamente disponibili per ogni attività da svolgere all'interno del computer, o dei computer, utilizzati. Prendendo in considerazione un ambiente cosiddetto distribuito, va da sè che i federati hanno bisogno di scambiarsi informazioni sul loro stato. Il Data Distribution Management (DDM) è lo strumento che permette l'armonizzazione operativa dei federati. [2]
Il DDM - servizio incluso in RTI - ha il compito di gestire la trasmissione dei dati e gli aggiornamenti dello stato che vengono inviati nelle varie simulazioni, con l'obiettivo di ottimizzare tutte le risorse disponibili, ovvero minimizzare il volume dei dati trasmesso sulla rete che collega tutti i federati tra loro.
Nella pratica, poi, è comune che un federato necessiti solamente di un sottoinsieme di tutti i dati messi a disposizione dal sistema. Ad esempio, in caso si possieda una federazione che simuli lo spostamento delle persone in una grande città, si avrà una federazione che simula quante persone vi sono in ogni quartiere della città. Un cittadino che deve dirigersi al supermercato, da un punto di vista logico, sarà interessato solamente alla quantità di persone presenti in quel luogo specifico: è irrilevante sapere il numero di persone negli altri luoghi della città. In altre parole, il DDM è un servizio che permette di velocizzare la selezione dei dati rilevanti durante un certo processo di elaborazione: il suo scopo, fra gli altri, è quello di istituire un filtraggio delle informazioni pertinente all'obiettivo che si prepone l'HLA.
Più in particolare, l'obiettivo di un DDM all'interno di un'HLA è quello di limitare la ricezione di messaggi - ossia di dati e informazioni - tra i federati di grandi federazioni distribuite a seconda dei loro interessi, ossia delle informazioni a loro pertinenti. Così da ridurre:

- La quantità di dati da processare da parte del federato che li riceve;
- Il traffico di dati sulla rete, così da ottimizzare il funzionamento generale dell'HLA.

Intuitivamente, dunque, programmare l'algoritmo che governa il servizio di DDM all'interno di un'HLA dovrebbe seguire un principio di operatività

semplice, chiaro e diretto. Specificatamente l'algoritmo di DDM dovrebbe attenersi a tre principi chiave [3][4][5] (Vedi Figura 2.2):

1. **Efficienza**: I servizi di DDM dovrebbero permettere un «overhead» minimo in termini computazionali, di latenza di messaggio e di utilizzo di memoria. Operazioni complesse come ad esempio quelle di «string comparison», dovrebbero essere evitate, quando possibile. In buona sostanza, ogni servizio deve giustificare i costi che comporta. Un servizio costoso può essere giustificabile solo se i benefici superano i costi.

2. **Scalabilità**: Ossia la capacità, in questo caso, di un gruppo software operanti in cluster di hardware di adattarsi elasticamente al carico di lavoro. Più specificatamente, la scalabilità qui si intende in termini di:

   - Complessità computazionale per gestire efficacemente le richieste;
   - Larghezza di banda per gestire efficacemente il traffico di messaggi nella distribuzione delle informazioni;
   - Memoria disponibile per gestire efficacemente la memorizzazione e il mantenimento di informazioni e dati rilevanti.

   I parametri che influenzano la scalabilità sono:

   - Il numero di federati che compongono la federazione;
   - Il numero di entità simulate in ogni singolo federato;
   - La complessità media che interessa ciascuna entità;
   - Il tasso di interazione tra i federati e le entità al loro interno;
   - La localizzazione degli oggetti a cui si riferiscono le entità dei federati;
   - Lo scenario in cui le simulazioni operano.

3. **Semplicità dell'interfaccia**: In parole povere, il DDM deve offrire la giusta interfaccia, quella ideale a filtrare i dati funzionali allo scopo delle federazioni che compongono l'HLA. L'interfaccia, in tal senso, dovrebbe essere intuitiva e chiara nell'utilizzo. La sua funzionalità è il risultato di un compromesso tra il filtraggio, i servizi forniti dall'RTI

e quelli operati dai federati. In generale, le interfacce con funzionalità limitate sono più semplici da utilizzare di quelle che impiegano funzionalità complesse e potenti. In buona sostanza, l'obiettivo è quello di predisporre un'interfaccia semplice che allo stesso tempo supporti la complessità delle operazioni di DDM necessarie all'interno dell'HLA.

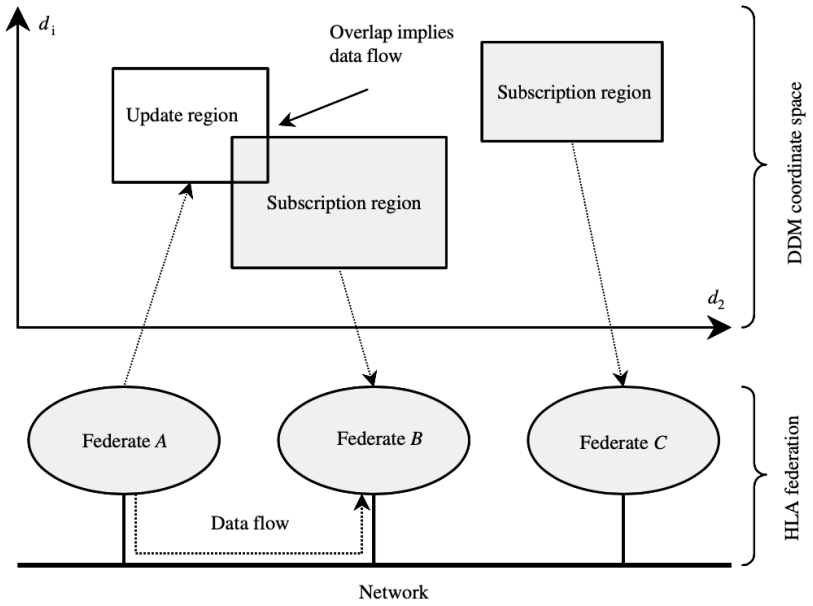

Figura 2.2: Esemplificazione concettuale di DDM. [2]

Nel DDM i federati vengono suddivisi in due gruppi [3][4][5]:

- «Publisher»: Quelli che pubblicano informazioni;
- «Subscriber»: Quelli che chiedono di ricevere informazioni.

Il DDM dovrà gestire il trasferimento in maniera ottimale per ridurre al massimo l'utilizzo della rete, mettendo in campo una serie di servizi atti ad evitare trasferimento di dati inutili che dovranno essere scartati in seguito.

Capitolo 3

Algoritmi del DDM

3.1 Introduzione al Matching

Il DDM svolge una serie di operazioni (o servizi) che servono a ridurre il quantitativo di dati non necessari nel trasferimento, così da diminuire drasticamente i due fattori fondamentali per il funzionamento del sistema [3]:

1. Il traffico sulla rete utilizzata;
2. Il carico di lavoro dei federati riceventi.

Diminuendo così il carico di dati di questi ultimi. In altre parole, il DDM si occupa di gestire la selezione delle informazioni rilevanti "a monte" del processo, così da snellire e velocizzare il carico di lavoro da svolgere "a valle". [3][4][5][6]

Le operazioni di trasferimento dei dati sono suddivise in cinque step:

1. Il Publisher dichiara i dati che produrrà e il Subscriber quelli da ricevere;
2. L'algoritmo che svolge le operazioni di «matching», filtra i vari federati che partecipano all'operazione di trasferimento, descrivendo l'elenco delle connessioni necessarie a svolgerla;
3. L'RTI apre le connessioni sopra citate;

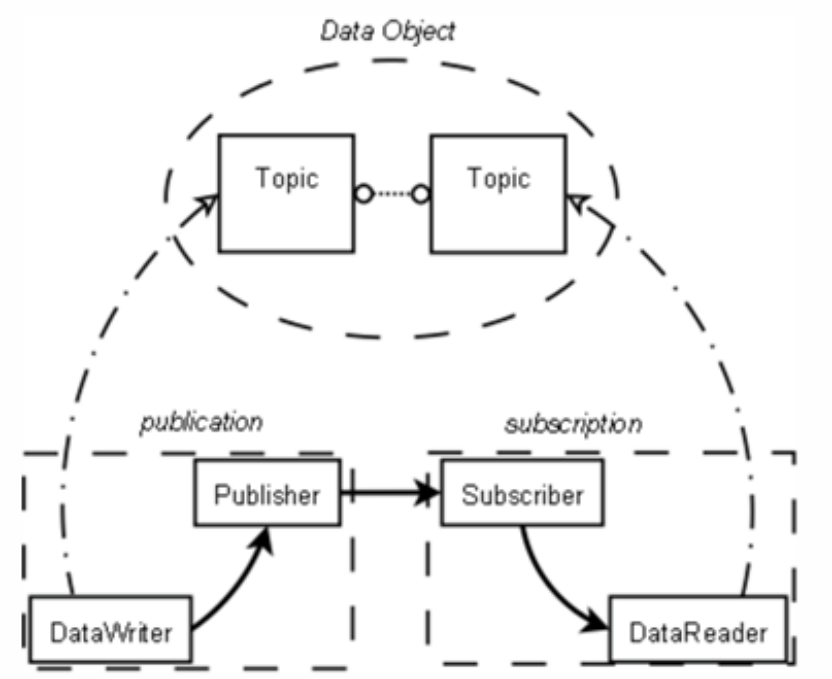

Figura 3.1: Una rappresentazione concettuale del rapporto tra federati Publisher e Subscriber. [opendds.org]

4. I Publisher trasmettono i dati ai Subscriber a cui sono stati connessi;
5. I Subscriber filtrano i dati ricevuti dai Publisher selezionando quelli utili, scartando di conseguenza le informazioni non pertinenti.

Da questo breve elenco, si nota che le operazioni di filtrazione sono due: al punto 2 e al punto 5. A prima vista, intuitivamente, il primo filtraggio potrebbe essere eliminato, così da ridurre i tempi di elaborazione. Approccio questo che, seppur logico e funzionale, comporta un trade-off in termini di carico di dati nell'operazione di scambio. Infatti, se P è il numero di Publisher ed S quello di Subscriber, il carico di scambio ammonterebbe a:

$$P * S$$

Ora, si supponga il caso esemplare di un bilanciamento fra il numero di publisher e quello di subscriber: è facile capire che aumenterebbe esponen-

zialmente - al quadrato - comportando la crisi computazionale del sistema. Anche con una leggera crescita dei federati, tuttavia, va notato che il numero di scambi risulterebbe essere decisamente limitato. L'operazione di «matching» al punto 2 - ossia la prima operazione di filtro e selezione - è dunque fondamentale: è necessaria a ridurre il numero di trasferimenti (ossia di connessioni) facendosi carico di una parte sostanziale del filtraggio dati prima del trasferimento.
Ecco che allora la fase in cui i publisher ed i subscriber si «iscrivono» - ossia dichiarano i dati che intendono produrre e ricevere - è fondamentale a rendere più efficiente lo scambio della fase 4, riducendo il carico del filtraggio in fase 5.
In particolare, si noti che l'algoritmo che si occupa delle operazioni di matching lavora entro un sistema spaziale di coordinate multidimensionali che viene denominato «routing space». Al suo interno prevede una serie di «region», sezioni di spazio formate da vari «extent», ovvero dei sottospazi di forma rettangolare all'interno delle «region». Le «region» prodotte dai publisher vengono denominate «update», mentre quelle richieste e inserite dai subscriber «subscription» [10][11][12].
Si consideri ora, a titolo esemplificativo, un routing space a due dimensioni (ad esempio: «Dimensione 1», «Dimensione 2») inscritte nel secondo quadrato di un sistema di riferimento cartesiano (vedi Figura 3.2). Nello spazio considerato, sono presenti due region («Region A» e «Region B») che a loro volta raggruppano una serie di extent («Extent...», «Extent...», etc...). Un sistema di questo tipo può rappresentare, nella realtà, delle persone o dei mezzi in movimento, dando la possibilità di simulare, a seconda degli input, dei dati prodotti dai publisher, una serie di azioni simulate da parte degli extent.

Le region considerate - «Region A» e «Region B» - comprendono rispettivamente due - «Extent A1» ed «Extent A2» - e tre - «Extent B1», «Extent B2» e «Extent B3» - extent. Si considerino ora le due region come contrapposte (quindi, in termini logici, con funzioni-obiettivo differenti): la sovrapposizione si registra tra A1 e B1 [11]. Le dimensioni ovviamente possono essere la rappresentazione di concetti spaziali come anche qualitativi: la posizione su un campo di battaglia, ma anche, ad esempio, la capacità di una guardia o di un controllore di rilevare una persona al buio. O ancora, simulando una pizzeria, la dimensione 1 potrebbe significare il tipo di servizio richiesto da un consumatore (ad esempio una bibita) in modo che questa

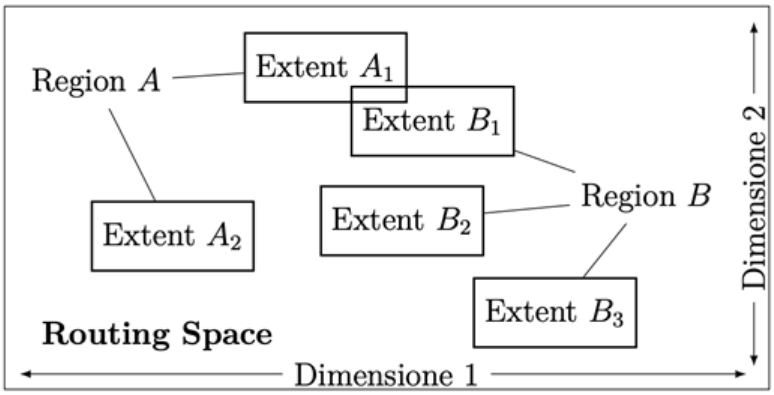

Figura 3.2: Esemplificazione schematica di un Routing-Space (elaborazione dell'autore).

arrivi al cameriere al banco e non al pizzaiolo, a cui devono essere recapitate solo le informazioni della dimensione 2 relative ai gusti della pizza.
Una precisazione: le informazioni che i federati devono trasferire non sono contenuti nel routing space: quest'ultimo infatti svolge anche la funzione di filtro in un DDM, ma non solo: esso, infatti, attribuisce durante il filtraggio le informazioni individuate dalle dimensioni stesse. Banalmente, l'ordine della pizza arriverà anche al cameriere per dargli modo di gestire l'ordinazione nel caso, ad esempio, il gusto scelto preveda l'aggiunta di condimenti dopo cottura, olio piccante o basilico fresco, a discrezione del consumatore.

3.1.1 Il Matching nel DDM

L'operazione di matching sopra descritta si discosta da quella che si svolge nel DDM in termini di complessità. In primis, si deve considerare che un publisher invia solamente dati e un subscriber li riceve; da ciò si deduce che è inutile confrontare degli extent appartenenti a federati della medesima tipologia.
Gli extent dunque vengono suddivisi in due tipi, seguendo la stessa logica applicata per le region, in modo da poterli distinguere adeguatamente, ossia:

1. «Update extent»: Appartengono alla stessa «update region»;

2. «Subscription extent»: Appartengono alla stessa «subscription region».

Dunque, l'algoritmo di matching DDM confronterà solo update extent con subscription extent [3][4][12].
Altro fattore fondamentale da considerare è che i dati saranno filtrati di nuovo dai subscriber: l'algoritmo, dunque, non deve presentare un tasso di fallibilità proibitivo; in altre parole, economicamente parlando, il matching - operazione di per sé onerosa - può essere alleggerito senza che la piccola percentuale di dati non necessari, che passa il filtro indenne, rallenti significativamente il processo di trasferimento. Anzi, spesse volte un depotenziamento dell'algoritmo di matching comporta una velocità di trasferimento sensibilmente maggiore e quindi un volume di trasferimenti di dati maggiore. [7][8]

3.2 Il Routing Space

Come anticipato, il routing space, da un punto di vista concettuale, altro non è che uno spazio definito da una serie di coordinate di due o più dimensioni spaziali. I federati, a loro volta, creano degli extent che indicano un'area delimitata all'interno di questo spazio dove invieranno o riceveranno i dati, a seconda che siano publisher o subscriber. Un extent, considerato uno spazio bidimensionale, ha forma rettangolare. Un gruppo di extent, a loro volta, da forma a una region, che può essere «update» o «subscription» region. Concettualmente, dunque, quando una subscription region viene sovrapposta ad un'update region, il subscriber che svolge la subscription riceverà i dati dal relativo publisher (vedi Figura 3.3).

Se si guarda all'esempio in Figura 3.3, si hanno tre update extent («U1», «U2», «U3») e tre subscription extent («S1», «S2», «S3») all'interno di un routing space bidimensionale. Si nota dunque che U1 non si sovrappone a nessun subscription extent e quindi non invierà dati durante la sessione individuata dal routing space. Al contrario, U2 è sovrapposto a S2 ed U3 a S3: in questo caso, U2 invierà i dati a S2 ed ugualmente U3 li invierà a S3. La sovrapposizione fra S2 e S3 non viene presa in considerazione dall'algoritmo e non avverrà nessun scambio di dati. Lo stesso si applica nel caso di sovrapposizione tra update extent.
Si supponga ad esempio una rete GPS di ricezione dati per smartphone. Le

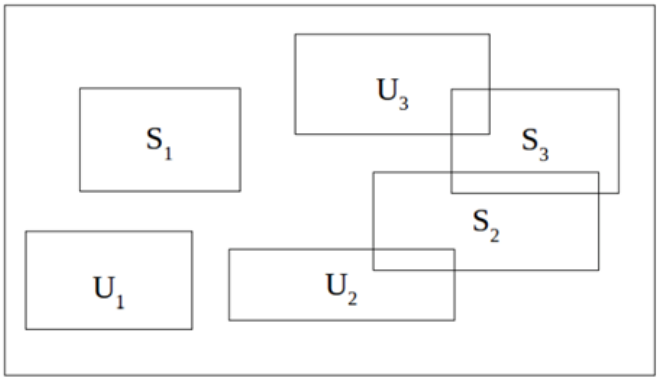

Figura 3.3: Update e Subscription Extent in un Routing Space bidimensionale (elaborazione dell'autore).

due dimensioni verranno utilizzate per determinare le coordinate spaziali dei dispositivi, delle antenne e dei satelliti, mentre la terza (ossia la dimensione del filtro) servirà a distinguere tra GPS e copertura della rete. In questo caso, gli smartphone saranno identificati come subscriber mentre le antenne e i satelliti come publisher. Gli update extent bidimensionali rappresenteranno l'area di copertura dei publisher mentre le subscription extent la forza del segnale dei subscriber. Quando un publisher si sovrappone ad un update il segnale renderà possibile la comunicazione fra la cellula all'interno dello smartphone ed il satellite.

3.3 Algoritmi di Matching nel DDM

Gli approcci con cui si impostano delle operazioni di matching si possono dividere in due categorie:

1. Approccio grid-based;

2. Approccio region-based.

Questi due non sono ovviamente mutualmente esclusivi, anzi: possono essere combinati per sfruttare e bilanciare i punti di forza e di debolezza

di ognuno. Nei seguenti sottoparagrafi si identificheranno il numero di publisher con «p» e il numero di subscriber con «s», mentre «d» indicherà il numero di dimensioni.

3.3.1 Parallel Brute Force

Il Parallel Brute Force è l'algoritmo di matching più semplice: un controllo sequenziale in ordine binario di ogni update extent con ognuno dei subscription extent di un database. La complessità computazionale «O» di questo algoritmo si può esprimere come quadratica [14]:

$$O(d * (m * n))$$

Da un punto di vista operativo, la sua semplicità e relativa economicità - i due punti di forza più evidenti dell'algoritmo - sono controbilanciati da una limitata scalabilità ed una lentezza dell'esecuzione. In caso vi siano molte sovrapposizioni, l'algoritmo ha la probabilità di segnalare delle intersezioni nella prima fase che è sensibilmente elevata. Soprattutto con database di dimensioni piccole e medie, specie se nei passaggi successivi si evitano di considerare gli extent appartenenti ad una stessa region, si possono avere vantaggi prestazionali considerevoli. Al crescere della dimensione del database, ovviamente, i punti deboli dell'algoritmo lo rendono poco utile. [12]

3.3.2 Parallel Grid-Based

Questo algoritmo di matching è ottimo per ridurre le risorse accessorie richieste per svolgere l'operazione computazionale in fase operativa da un algoritmo brute force. Applicare l'approccio parallel grid-based permette di tagliare i costi di matching di tutte le region attraverso una partizione del routing space sfruttando una griglia di celle [9]. L'operazione sostanziamente è una mappatura degli extent, siano essi update o subscriptions: se appartengono alla stessa cella verranno considerati sovrapposti, anche se non lo sono effettivamente. Questo, com'è ovvio, permette di applicare una soluzione di scalabilità semplice che bilancia la sequenzialità del brute force. D'altro canto, tuttavia, la sovrapposizione di publisher e subscriber della stessa cella non interessata aumenta il margine di dati non necessari, di conseguenza le risorse necessarie e la potenza di calcolo richiesta nella

fase di filtraggio da parte dei subscriber stessi.
Chiaramente, una variabile cruciale è la dimensione delle celle: stante il fatto che un extent che occupa due o più celle viene elaborato come appartenente ad ognuna di esse, la loro dimensione è inversamente proporzionale ai costi di gestione delle operazioni di mappatura, specie nel caso di simulazioni particolarmente dinamiche. Altresì, più queste saranno grandi più il carico di dati irrilevanti da filtrare attraverso i subscriber sarà elevato. [15] Molte ricerche, in questo senso, sono state dedicate nel corso degli anni a determinare un range di valori che ottimizzino il parametro della dimensione delle celle, sviluppando algoritmi che producano griglie a dimensione variabile che si adattino alla distribuzione spaziale degli extent, sia nello spazio che nelle iterazioni. [15] Chiaramente, costi ed efficienza del filtraggio di un algoritmo basato su un approccio parallel grid dipendono anche dalla larghezza della banda, dalla memoria disponibile, dalla velocità di calcolo e altre variabili, che rendono difficile ed in sostanza troppo oneroso, determinare un range di valori sufficientemente preciso. [10]

3.3.3 Sort-Based

Il vantaggio principale di questo approccio è la riduzione del numero di confronti necessari a ordinare i bordi degli extent (che, si ricordi, hanno forma rettangolare). [14] Questo metodo lavora su una dimensione spaziale del routing space alla volta, proiettando i confini degli extent dentro un singolo vettore dimensionale (vedi Figura 3.4). Un extent viene considerato in «match» con un altro solamente quando le due dimensioni che lo caratterizzano nel routing space combaciano con quelle di un altro. Coerentemente, l'approccio sort-based crea un vettore che contiene una delle due dimensioni relative ad ogni extent, che vengono sistematizzate sequenzialmente.

Com'è ovvio, questo approccio ha il vantaggio che, secondo l'ordine sequenziale, un extent può solo precedere, includere o seguirne un altro, rendendo palesi le sovrapposizioni. Altresì, si possono ordinare gli extent secondo due liste così denominate:

- «SubscriptionSetBefore»: Contiene tutti i subscription extent i cui confini sono già usciti dal vettore;
- «SubscriptionSetAfter»: Contiene tutti i subscription extent i cui confini devono ancora entrare nel vettore.

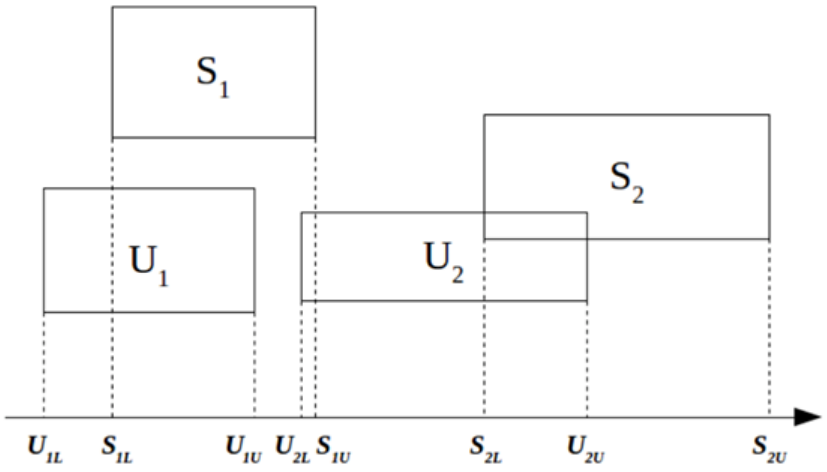

Figura 3.4: Proiezione delle dimensioni degli extent di U e S su un vettore sequenziale (elaborazione dell'autore).

Dunque, estraendo una delle dimensioni spaziali di un extent, si possono trarre delle conclusioni analizzando lo stato dello stesso nelle due liste: se si registra il limite inferiore, l'extent non si sovrappone a nessuno dei subscriber presenti nella lista «SubscriptionSetBefore»; al contrario, se si registra il limite superiore della dimensione spaziale, l'extent non si sovrappone a tutti i subscriber della lista «SubscriptionSetAfter».
Una scannerizzazione dei vettori dimensionali rappresenta un costo lineare: dunque la complessità computazionale dell'approccio sort-based dipende dalla complessità della sistematizzazione implementata. Se, ad esempio, si considera la Figura 3.4, dove si hanno due update extent («U1» e «U2») e i due subscription extent («S1» e «S2»), ipotizzando un algoritmo «linearithmic» che computa il numero di dimensioni spaziali degli extent, la complessità computazionale «O» è data da [14]:

$$O(d * ((p + s) * log(p + s)))$$

Il vantaggio principale di questo approccio rispetto a quello brute force e, relativamente, al grid-based, è la maggiore scalabilità del processo di computazione. Tuttavia, dei tre, è quello più "pesante" in termini di memoria,

dato che i vettori addizionali generati devono essere conservati durante le operazioni.
Va segnalato, infine, che sono stati analizzati due tipi di algoritmi sort-based, i «serial» ed i «parallel». L'algoritmo «sort-based serial», come si intuisce dalla denominazione, svolge le operazioni in serie, ossia una dopo l'altra, col vantaggio che nessuna informazione processata venga perduta, a discapito di un dispendio di tempo non indifferente; nel «sort-based parallel», invece, le operazioni vengono svolte in parallelo, con ovvi vantaggi in termini di tempistiche e prestazioni in termini di velocità bilanciati da una ridotta precisione.

3.3.4 Sort-Based - Parallel con CUDA

Acronimo per «Compute Unified Device Architecture», l'architettura CUDA è una soluzione per migliorare le prestazioni dei DDM nella gestione delle cosiddette infrastrutture «run-time» (RTI) nelle HLA. In particolare, questo approccio viene applicato nei casi in cui le operazioni di matching si rivelino particolarmente laboriose [15]:

«When a significant number of regions are involved in a simulation, it takes a considerable amount of time to perform region matching. Furthermore, performing computation intensive region matching affects the time to perform other managements in RTI, e.g. Object Management and Time Management. Thus it could become the performance bottleneck of the whole RTI system.»

Per risolvere questo problema prestazionale, quindi economicizzare i processi, si opera un matching delle region in parallelo demandandone l'esecuzione a una «graphical processing unit» («GPU»). Il vantaggio di questo sistema è quello di poter combinare gli approcci descritti nel paragrafo, sfruttandone i punti di forza, contenendo l'utilizzo di memoria e quindi economicizzando e velocizzando l'intero processo. Nel prossimo capitolo, si discuterà l'ottimizzazione dell'algoritmo con approccio Parallel CUDA.

Capitolo 4

Ottimizzazioni dell'Algoritmo Sort-Based Parallel in CUDA

L'applicazione oggetto del presente capitolo ha come obiettivo la modifica dell'algoritmo «parallel sort» per un'applicazione in GPU-CUDA. Nel primo paragrafo si introducono le applicazioni GPU-CUDA; nel secondo si svolge un commento passo a passo delle tre parti in cui è stato suddiviso il codice che implementa l'algoritmo; nel terzo si opera una discussione dell'adattamento dell'algoritmo presentato per un'applicazione su GPU-CUDA.

4.1 Introduzione GPU e CUDA

In buona sostanza, l'architettura CUDA offre un modello di programmazione per eseguire operazioni di carattere generale sulla GPU che risponde alla necessità di ridurre drasticamente il tempo dell'esecuzione runtime di molti algoritmi - come quelli visti nei sotto paragrafi del capitolo precedente - impiegando hardware relativamente poco costoso.
La GPU, nata come un dispositivo altamente specializzato, si è evoluta negli ultimi decenni in un «highly parallel, multithreaded, many-core processor with tremendous computational horsepower and high memory bandwidth». [16]
In particolare, le GPU sviluppate da NVIDIA per applicazioni CUDA forniscono un'interfaccia C in cui la curva di apprendimento è decisamente bassa: in altre parole, facile da apprendere e da utilizzare. Le GPU-CUDA,

grazie a ciò, negli ultimi vent'anni, hanno permesso di risolvere problemi computazionali particolarmente complessi sempre più velocemente. Inoltre, l'emergere delle architetture a GPU multiple ha permesso alle applicazioni CUDA di risolvere problemi di complessità prima inavvicinabile, specie quando la memoria richiesta per l'elaborazione superava quella disponibile in una singola GPU. In parole povere, combinando la capacità di calcolo di diverse GPU in contemporanea permette di aumentare drasticamente la velocità e l'efficienza del processo computazionale [16].
Storicamente, infatti [17]:

«Performance scaling of single-thread processors stopped in 2002 and has fueled the use of multicore Graphics Processing Units (GPUs) which have been growing in transistor count by 65% annually. The current generation of NVIDIA's Fermi GPU consists of 512 cores and is capable of executing 24,576 concurrent thread kernels for efficient stream processing. GPUs [...] in the year 2015, use 11nm technology and contain around 5,000 cores, which should render them capable of around 20 Teraflops [...]. In the past few years, the GPU has evolved into an increasingly convincing computational platform for non-graphics applications [...]»

Oggi le applicazioni GPU-CUDA sono innumerevoli e vengono utilizzate per stime in tempo reale o predittive nelle scienze meteorologiche o in applicazioni mediche e militari. Anche nei più recenti studi di gestione del traffico a livello nazionale e internazionale, nonché nelle operazioni di trading elettronico.
Come si può notare, queste applicazioni hanno in comune, in termini di dati da processare, carichi di lavoro importanti, tant'è che le applicazioni in tempo reale spesso si rivelano troppo precise e, di conseguenza, troppo costose per essere effettivamente ed economicamente utili. Ecco che una stima approssimativa ma abbastanza affidabile elaborata entro tempistiche accettabili si rivela decisamente più utile di una più precisa ma datata. Gli algoritmi implementati, di conseguenza, devono essere in grado di adattarsi e allocare le risorse appropriate a seconda degli obiettivi e dei carichi di lavoro, valutando i progressi in funzione del tempo di esecuzione ed offrendo risultati intermedi in tempo reale con una approssimazione accettabile. [17]

L'architettura di una GPU funziona secondo un linguaggio di programazione CUDA, estensione proprietaria di quello C e C++ (vedi Figura 4.1).

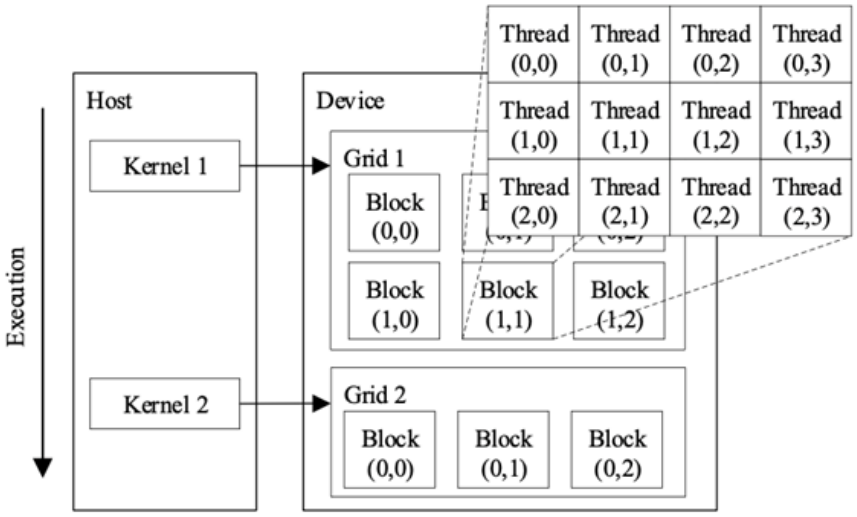

Figura 4.1: Modello di esecuzione di una Architettura CUDA. [17]

Il flusso di esecuzione del programma è controllato dalla CPU (l'«host») mentre i calcoli più impegnativi vengono affidati alla GPU (il «device») dove vengono utilizzate funzioni parallele definite «kernels», ossia una routine, il corrispettivo degli «inner loops» nelle implementazioni degli algoritmi nei linguaggi tradizionali. In altre parole, operazioni di coding passate attraverso iteratori interni. La GPU lancia una griglia di «threads» organizzati in blocchi e mappati. I thread appartenenti agli stessi blocchi possono essere sincronizzati condividendo gli stessi dati nella memoria condivisa a bassa latenza («low latency shared memory»). È possibile anche attivare processi di sincronizzazione e collaborazione fra differenti blocchi di thread "splittando" il programma in kernels multipli utilizzando la memoria globale per salvare i dati fra le varie routine (vedi Figura 4.2). [17]

Infine, va notato che [18]:

«To port a software program to the GPU, each sequential operation must be analyzed and replaced with a parallel primitive that execute the

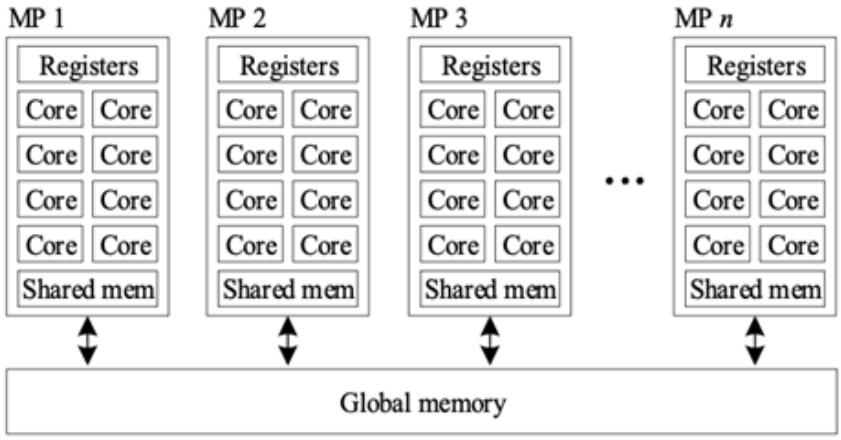

Figura 4.2: Architettura tipica di una GPU [17].

same work, but in fewer steps.»

In sostanza, dunque, le applicazioni sviluppate per GPU devono sfruttare processi di parallelismo di grande portata per beneficiare appieno delle enormi architetture parallele del processore.
In conclusione, si desidera sottolineare che nell'applicazione oggetto di questa tesi si è fatto largo uso di «Thrust», una libreria di algoritmi paralleli che ricordano il linguaggio C++ della «Standard Template Library» («STL»). La libreria, grazie alla sua interfaccia di alto livello, ha permesso di migliorare significativamente la produttività e l'efficienza nelle operazioni di programmazione, specie per quanto riguarda la GPU: «Thrust». Tra le altre tecnologie considerate, si propone di migliorare l'interoperabilità dell'architettura CUDA nello sviluppo di applicazioni ad alte prestazioni.

4.2 Modifiche all'Algoritmo Parallel Sort

L'applicazione oggetto del presente capitolo, come si è visto, ha come obiettivo la modifica dell'algoritmo «parallel sort» per un'applicazione in GPU-CUDA. In questo paragrafo, si commentano i vari passaggi salienti del codice

di partenza (sfruttando anche i commenti autoesplicativi inseriti nel codice). [19]
Anzitutto, si è svolto un settaggio preliminare delle variabili e dei vettori utili a selezionare e salvare in memoria i dati che verranno elaborati in seguito:

```
#include <vector>
#include <cassert>
#include <iostream>
#include <algorithm>
#include <parallel/algorithm>
#include <omp.h>

#include "omp_timer.hh"
#include "interval.h"c
#include "parallel_sort_matching.hh"
#include "sort_matching_common.hh"

#if 0
#define BEGIN_TIMER(n) {
omp_timer my_timer(n);
#define END_TIMER()
}
#else
#define BEGIN_TIMER(n)
#define END_TIMER()
#endif
```

In seguito, si svolge un settaggio generale delle dimensioni e dei vincoli da considerare durante lo svolgimento del sort matching, nonché degli intervalli da considerare durante le subscription:

```
/* Parallel sort-based matching algorithm */
extern "C"
size_t parallel_sort_matching(const struct interval* sub, size_t n,
                              const struct interval* upd, size_t m)
{
    omp_timer sbm_timer("Parallel SBM");
```

```
    const size_t n_endpoints = 2*(n+m);
    std::vector<endpoint> endpoints(n_endpoints);
    size_t nmatches = 0;
```

Si procede, dunque, ad impostare la sezione di codice parallelo che assegnerà i valori degli endpoint ai rispettivi vettori.
Lo «shared», in questo caso, descrive le variabili che saranno condivise da tutti i thread così da permettere un'esecuzione in parallelo fluida e veloce:

```
//Push subscription intervals into the endpoints array (in parallel)
    BEGIN_TIMER("Fill");
#pragma omp parallel default(none) shared(n,m,sub,upd,endpoints)
    {
#pragma omp for
        for (size_t i=0; i<n; i++) {
            assert(sub[i].lower < sub[i].upper);
            endpoints[i] = endpoint(i, sub[i].lower,
                        endpoint::LOWER, endpoint::SUBSCRIPTION);
            endpoints[i+n] = endpoint(i, sub[i].upper,
                        endpoint::UPPER, endpoint::SUBSCRIPTION);
        }
#pragma omp for
        for (size_t i=0; i<m; i++) {
            assert(upd[i].lower < upd[i].upper);
            endpoints[2*n+i] = endpoint(i, upd[i].lower,
                        endpoint::LOWER, endpoint::UPDATE);
            endpoints[2*n+m+i] = endpoint(i, upd[i].upper,
                        endpoint::UPPER, endpoint::UPDATE);
        }
    }
    // end parallel region
    END_TIMER();
```

Si procede riordinando tutti gli endpoint ottenuti all'interno del vettore risultante:

```
    //Sort endpoints (in parallel)
    BEGIN_TIMER("Sort");
```

```
    __gnu_parallel::sort(endpoints.begin(), endpoints.end());
    END_TIMER();
```

Il «timer», in questo caso, ha finalità di controllo delle prestazioni; in altre parole, sarà utilizzato per capire quanto tempo impiegherà l'algoritmo a ordinare i dati corrispondenti.
Dunque, si prosegue con l'operazione di riduzione [21]:

```
    BEGIN_TIMER("Reduction");
//added last 2 variables to remove compilation error
#pragma omp parallel reduction(+:nmatches) default(none)
                shared(sub_insert,sub_delete,upd_insert,
                        upd_delete,subscriptions,updates,
                        endpoints,n_endpoints)
    {
        const int my_rank = omp_get_thread_num();
        const int thread_count = omp_get_num_threads();
        const size_t local_n =
            (n_endpoints + thread_count - 1) / thread_count;
        const size_t local_idx_start = my_rank*local_n;
        const size_t local_idx_end =
            std::min(local_idx_start + local_n, n_endpoints);
// First step: local scans in parallel
#ifndef NDEBUG
#pragma omp critical
        std::cout << "Thread " << my_rank << " entering first block"
                  << " local_idx_start=" << local_idx_start
                  << " local_idx_end=" << local_idx_end
                  << std::endl;
#endif
        {
/* Start a new scope so that the local variables
        below will disappear when no longer needed */
            epset& my_sub_insert( sub_insert[my_rank] );
            epset& my_sub_delete( sub_delete[my_rank] );
            epset& my_upd_insert( upd_insert[my_rank] );
            epset& my_upd_delete( upd_delete[my_rank] );
```

```
            for (size_t idx = local_idx_start;
                                idx < local_idx_end; idx++) {
                const endpoint& my_ep( endpoints[idx] );
                if (my_ep.t == endpoint::SUBSCRIPTION) {
                    if (my_ep.e == endpoint::LOWER) {
                        my_sub_insert.insert(my_ep.id);
                    } else {
                        if ( my_sub_insert.find(my_ep.id) )
                            my_sub_insert.remove(my_ep.id);
                        else
                            my_sub_delete.insert(my_ep.id);
                    }
                } else {
                    if (my_ep.e == endpoint::LOWER) {
                        my_upd_insert.insert(my_ep.id);
                    } else {
                        if ( my_upd_insert.find(my_ep.id) )
                            my_upd_insert.remove(my_ep.id);
                        else
                            my_upd_delete.insert(my_ep.id);
                    }
                }
            }
#ifndef NDEBUG
#pragma omp critical
    std::cout << "Thread " << my_rank << " exiting first block"
        << " subs_insert=" << sub_insert[my_rank].size()
        << " subs_delete=" << sub_delete[my_rank].size()
        << std::endl;
#endif
        }
#pragma omp barrier
```

Questa parte di codice, eseguita dai vari thread del sistema, si occupa dell'esecuzione dell'operazione di scan in parallelo [20].
Ogni thread si ritroverà con una parte di vettore su cui effettuare l'operazione.

```
//Second step: global scan (master only)
#pragma omp master
    {
        for (int p=1; p<thread_count; p++) {
            subscriptions[p]
                .merge( subscriptions[p-1] )
                .merge( sub_insert[p-1] )
                .subtract( sub_delete[p-1] );

            updates[p]
                .merge( updates[p-1] )
                .merge( upd_insert[p-1] )
               .subtract( upd_delete[p-1] );
#ifndef NDEBUG
        std::cout << "p=" << p
        << " subs=" << subscriptions[p].size()
        << " upd=" << updates[p].size() << std::endl;
#endif
        }
    }
```

Infine, il thread master si occuperà dell'unione dei dati di tutti i thread preservando l'ordinamento. Questo passaggio, si noti, è cruciale per il corretto svolgimento delle operazioni. È presente un comando per visualizzare in output i dati ed eseguire un debug durante il processo di scan, così da effettuare un ulteriore controllo sui risultati intermedi e, eventualmente, rilevare degli errori.
Terminata questa parte dell'algoritmo, segue la terza; ossia la fase in cui si eseguirà un ciclo per assegnare o rimuovere i dati negli appositi vettori, effettuando un controllo per verificare che siano parte degli endpoint LOWER o UPPER.

```
#pragma omp barrier
// Third step: local scans (in parallel)
    {
//give shorter names to subscriptions[my_rank] and updates[my_rank]
        epset& my_sub( subscriptions[my_rank] );
        epset& my_upd( updates[my_rank] );
```

```
            for (size_t idx = local_idx_start;
                                    idx < local_idx_end; idx++) {
            const endpoint& my_ep = endpoints[idx];
                if (my_ep.t == endpoint::SUBSCRIPTION) {
                if (my_ep.e == endpoint::LOWER) {
                    // std::cout << "S I " << my_ep.id << std::endl;
                    my_sub.insert(my_ep.id);
                } else {
                    // std::cout << "S R " << my_ep.id << std::endl;
                    my_sub.remove(my_ep.id);
                    nmatches += my_upd.size();
                }
            } else {
                if (my_ep.e == endpoint::LOWER) {
                    // std::cout << "I R " << my_ep.id << std::endl;
                    my_upd.insert(my_ep.id);
                } else {
                    // std::cout << "I R " << my_ep.id << std::endl;
                    my_upd.remove(my_ep.id);
                    nmatches += my_sub.size();
                }
            }
        }
        assert( my_sub.empty() );
        assert( my_upd.empty() );
    }
}
//end parallel region
END_TIMER();
return nmatches;
}
```

Successivamente si procede con il codice che descrive i passaggi di sort matching seriale, dove si impostano le dimensioni delle variabili, gli endpoints, nonché le specifiche riguardanti il timer di controllo, le coordinate degli endpoint delle subscription e le dimensioni corrispondenti degli update. Segue l'ultima fase in cui, grazie alla funzione «std::sort» si procede con l'ordinamento degli endpoint fra update e subscriber corrispondenti:

```
/* Serial sort-based matching algorithm */
extern "C"
size_t sort_matching( const struct interval* sub, size_t n,
                      const struct interval* upd, size_t m )
{
    omp_timer sbm_timer("Serial SBM");
    const size_t n_endpoints = 2*(n+m);
    std::vector<endpoint> endpoints(n_endpoints);
    size_t nmatches = 0;

    //Push subscription intervals into the endpoints array
    BEGIN_TIMER("Fill");
    for (size_t i=0; i<n; i++) {
        assert(sub[i].lower < sub[i].upper);
        endpoints[i] = endpoint(i, sub[i].lower,
                                   endpoint::LOWER,
                                   endpoint::SUBSCRIPTION);
        endpoints[i+n] = endpoint(i, sub[i].upper,
                                   endpoint::UPPER,
                                   endpoint::SUBSCRIPTION);
    }

    for (size_t i=0; i<m; i++) {
        assert(upd[i].lower < upd[i].upper);
        endpoints[2*n+i] = endpoint(i, upd[i].lower,
                                   endpoint::LOWER,
                                   endpoint::UPDATE);
        endpoints[2*n+m+i] = endpoint(i, upd[i].upper,
                                   endpoint::UPPER,
                                   endpoint::UPDATE);
    }
    END_TIMER();

    //Sort endpoints
    BEGIN_TIMER("Sort");
    std::sort(endpoints.begin(), endpoints.end());
    END_TIMER();
```

```
    epset subscriptions(n);
    epset updates(m);

    BEGIN_TIMER("Scan");
    for (size_t idx = 0; idx < endpoints.size(); idx++) {
        const endpoint& ep = endpoints[idx];
        if (ep.t == endpoint::SUBSCRIPTION) {
            if (ep.e == endpoint::LOWER) {
                //std::cout << "S I " << ep.id << std::endl;
                subscriptions.insert(ep.id);
            } else {
                //std::cout << "S R " << ep.id << std::endl;
                subscriptions.remove(ep.id);
                nmatches += updates.size();
            }
        } else {
            if (ep.e == endpoint::LOWER) {
                //std::cout << "I R " << my_ep.id << std::endl;
                updates.insert(ep.id);
            } else {
                //std::cout << "I R " << my_ep.id << std::endl;
                updates.remove(ep.id);
                nmatches += subscriptions.size();
            }
        }
    }
    assert( subscriptions.empty() );
    assert( updates.empty() );
    END_TIMER();
    return nmatches;
}
```

Si conclude con l'«end» del timer di controllo delle prestazioni e il comando di «return nmatches» per la consegna dell'output al chiamante.

4.3 Adattamento dell'Algoritmo alla GPU

In buona sostanza, si può dire che le sezioni «parallel» del codice andavano modificate, riadattate e quindi inserite in un algoritmo capace di fornire le istruzioni di esecuzione corrette alla GPU. In questo senso - lo si è notato in chiusura al primo paragrafo e lo si sottolinea di nuovo qui -, i materiali a disposizione nella libreria Thrust sono stati fondamentali. Si è riadattato anche il main dell'algoritmo, così da poter eseguire un test ed un'esecuzione di prova del nuovo algoritmo sulla GPU. In particolare, si sono aggiunte le variabili necessarie all'esecuzione dell'algoritmo descritto nel precedente paragrafo su una GPU-CUDA. Dato che si sono riscontrati dei problemi nuovi con la versione "recente" di GPU-CUDA utilizzata, si è iniziato con un check ed adattamento del programma per verificare che la dichiarazione delle variabili nella parte di codice «shared» sia corretta e si è optato per un ulteriore controllo sul corretto funzionamento dell'architettura CUDA. Dunque, si sono adattate le dichiarazioni delle variabili per essere lette ed eseguite correttamente in base alla versione presente sulla macchina ed, allo stesso tempo, renderle il più leggibili e comprensibili possibile. Il codice è stato riorganizzato secondo i tre step originali analizzati nel paragrafo precedente ma riadattati per CUDA. Le tre parti di codice sono state migliorate in seguito ad una serie di prove in cui le loro performance sono risultate decisamente scadenti. Per evidenziare il miglioramento delle prestazioni, tempi di esecuzione ed eventuali log, si fa riferimento al capitolo successivo. Infine, si segnala che ad ogni operazione svolta dall'architettura CUDA, viene effettuato un check di verifica per eventuali errori e malfunzionamenti denominato «checkCuda(cudaGetLastError())» che ritorna un codice di errore in caso qualche operazione dell'architettura CUDA non vada a buon fine (implementazione qui sotto).

4.4 Dettagli Implementativi

In questo paragrafo si svolge una descrizione dei principali dettagli implementati dell'algoritmo realizzato e del suo adattamento all'applicazione GPU-CUDA. Dopo una prima impostazione dell'ambiente e delle variabili, il file apre con un controllo di eventuali errori dell'esecuzione su CUDA:

```
/* Parallel sort-based matching algorithm */
```

```
inline cudaError_t checkCuda(cudaError_t result)
{
  if (result != cudaSuccess) {
    fprintf(stderr, "CUDA Runtime Error: %s\n",
                        cudaGetErrorString(result));
    assert(result == cudaSuccess);
  }
  return result;
}
```

Successivamente si procede con la copia dei dati delle sottoscrizioni negli endpoint. In questa fase tutte le variabili sono nella memoria della GPU. Si imposta il numero totale dei thread che verranno gestiti dal kernel, si divide e si calcola l'intervallo di inizio e fine degli indici gestiti dal thread (ponendo particolare attenzione all'ultimo intervallo) e si esegue la sottoscrizione. In questa fase si esegue ulteriormente un controllo degli intervalli, verificando la loro validità e si inseriscono i dati nel vettore degli endpoint.

```
__global__ void cuda_copy_subscription(int n,
    struct endpoint* endpoints,
    struct interval* sub)
{
    //Inserisci gli intervalli di sottoscrizione nell'array
    //degli endpoint : tutte le variabili sono nella memoria della GPU
    //Numero totale di thread nel kernel
    int num_threads=blockDim.x*gridDim.x;
    //printf("num_threads %d\n", num_threads);

    //Divide e calcola inizio e fine per questo thread
    int n_local=(n+num_threads-1)/num_threads;
    int n_start=n_local*(blockIdx.x*blockDim.x + threadIdx.x);
    int n_end=n_start+n_local;

    //per l'ultimo intervallo
    if (n_end>n) {n_end=n;}

    //prima la sottoscrizione
    for (size_t i=n_start; i<n_end; i++) {
```

```
     //controlla che l'intervallo sia corretto, ovvero
     //l'inferiore e' minore del superiore
        assert(static_cast<struct interval>(sub[i]).lower <
        static_cast<struct interval>(sub[i]).upper);
        //inizializzo gli endpoint con l'inferiore
        endpoints[i] = endpoint(i,
            static_cast<struct interval>(sub[i]).lower,
            endpoint::LOWER, endpoint::SUBSCRIPTION);
        //inizializzo gli endpoint con il superiore
        endpoints[i+n] = endpoint(i,
            static_cast<struct interval>(sub[i]).upper,
            endpoint::UPPER, endpoint::SUBSCRIPTION);
    }
}
```

Si procede con la copia dei dati degli aggiornamenti negli endpoint. In questa fase le operazioni sono le stesse descritte precedentemente.

```
__global__ void cuda_copy_updates(int n, int m,
    struct endpoint* endpoints, struct interval* upd)
{
    //Numero totale di thread nel kernel
    int num_threads=blockDim.x*gridDim.x;
    //printf("num_threads %d\n", num_threads);

    //Divide e calcola inizio e fine per questo thread
    int m_local=(m+num_threads-1)/num_threads;
    int m_start=m_local*(blockIdx.x*blockDim.x + threadIdx.x);
    int m_end=m_start+m_local;
    //printf ("n %d n_local %d, n_start %d, n_end %d\n",n,
    //n_local, n_start, n_end);

    //per l'ultimo intervallo
    if (m_end>m) {m_end=m;}

    for (size_t i=m_start; i<m_end; i++) {
        //controlla che l'intervallo sia corretto, ovvero
        //l'inferiore e' minore del superiore
```

```
        assert(static_cast<struct interval>(upd[i]).lower <
        static_cast<struct interval>(upd[i]).upper);
        //inizializzo gli endpoint con l'inferiore
        endpoints[2*n+i] = endpoint(i,
            static_cast<struct interval>(upd[i]).lower,
            endpoint::LOWER, endpoint::UPDATE);
        //inizializzo gli endpoint con il superiore
        endpoints[2*n+m+i] = endpoint(i,
            static_cast<struct interval>(upd[i]).upper,
            endpoint::UPPER, endpoint::UPDATE);
        }
}
```

A questo punto si passa all'esecuzione del primo step dell'algoritmo di riferimento (Sort-Based) eseguito parallelamente ed adattato all'esecuzione sull'architettura CUDA. In questa prima fase, descritta dettagliatamente nel capitolo precedente, si è optato per il semplice adattamento dell'algoritmo all'esecuzione su GPU. Dopo l'analisi delle prestazioni, si è optato per il miglioramento dei singoli step per incrementarne le prestazioni:

```
__global__ void cuda_pass_1(int n_endpoints, int n, int m,
            class epset **sub_insert, class epset **sub_delete,
            class epset **upd_insert, class epset **upd_delete,
            struct endpoint *endpoints)
{
    //Numero totale di thread nel kernel
    int thread_count=blockDim.x*gridDim.x;
    int my_rank=blockIdx.x*blockDim.x + threadIdx.x;

    epset *my_sub_insert=sub_insert[my_rank]=new epset(n);
    epset *my_sub_delete=sub_delete[my_rank]=new epset(n);
    epset *my_upd_insert=upd_insert[my_rank]=new epset(m);
    epset *my_upd_delete=upd_delete[my_rank]=new epset(m);

    //Divide e calcola inizio e fine per questo thread
    const int local_n =
        (n_endpoints + thread_count - 1) / thread_count;
    const int local_idx_start = my_rank*local_n;
```

```
    int local_idx_end = local_idx_start + local_n;
    if (local_idx_end>n_endpoints) {local_idx_end=n_endpoints;}

    for (size_t idx = local_idx_start; idx < local_idx_end; idx++) {
        struct endpoint& my_ep= endpoints[idx];

        if (my_ep.t == endpoint::SUBSCRIPTION) {
if (my_ep.e == endpoint::LOWER) {
                my_sub_insert->insert(my_ep.id);
//printf ("sub_insert inserted id %lu\n", my_ep.id);
            } else {
                if ( my_sub_insert->find(my_ep.id) ) {
                    my_sub_insert->remove(my_ep.id);
//printf ("sub_insert removed id %lu\n", my_ep.id);
                } else {
                    my_sub_delete->insert(my_ep.id);
//printf ("sub_delete inserted id %lu\n", my_ep.id);
                }
            }
        } else {
            if (my_ep.e == endpoint::LOWER) {
                my_upd_insert->insert(my_ep.id);
//printf ("upd_insert inserted id %lu\n", my_ep.id);
            } else {
                if ( my_upd_insert->find(my_ep.id) ) {
                    my_upd_insert->remove(my_ep.id);
//printf ("upd_insert removed id %lu\n", my_ep.id);
} else {
                    my_upd_delete->insert(my_ep.id);
//printf ("upd_delete inserted id %lu\n", my_ep.id);
}
            }
        }
    }
}
```

Successivamente, si passa allo step 2 dell'algoritmo in cui vengono uniti e sottratti dal vettore delle sottoscrizioni i task paralleli dello step 1 per

ottenere un singolo vettore di sottoscrizioni e di aggiornamenti.

```
__global__ void cuda_pass_2(class epset **sub_insert,
            class epset **sub_delete, class epset **upd_insert,
            class epset **upd_delete, class epset **subscriptions,
            class epset **updates, int thread_count, int n, int m)
{
        subscriptions[0]=new epset(n);
        updates[0]=new epset(m);
            for (int p=1; p<thread_count; p++) {
subscriptions[p]=new epset(n);
updates[p]=new epset(m);
                (*subscriptions[p])
                    .merge( *subscriptions[p-1] )
                    .merge( *sub_insert[p-1] )
                    .subtract( *sub_delete[p-1] );

                (*updates[p])
                    .merge( *updates[p-1] )
                    .merge( *upd_insert[p-1] )
                    .subtract( *upd_delete[p-1] );
            }
}
```

Dunque, si procede con l'adattamento dello step 3 dell'algoritmo. La sua struttura, in questa prima scrittura, è la medesima del sort. Si è eseguito solamente l'adattamento del codice all'esecuzione su GPU:

```
__global__ void cuda_pass_3(int n_endpoints,
    class epset **subscriptions, class epset **updates,
    struct endpoint* endpoints, size_t *d_nmatches)
{
    //Numero totale di thread nel kernel
    int thread_count=blockDim.x*gridDim.x;
    int my_rank=blockIdx.x*blockDim.x + threadIdx.x;
//printf ("thread %d , n_endpoints %d, n %d m %d\n",my_rank,n_endpoints, n,

epset& my_sub( *subscriptions[my_rank] );
```

```
    epset& my_upd( *updates[my_rank] );

//Divide e calcola inizio e fine per questo thread
    size_t nmatches=0;
    const int local_n =
                (n_endpoints + thread_count - 1) / thread_count;
    const int local_idx_start = my_rank*local_n;
    int local_idx_end = local_idx_start + local_n;
    if (local_idx_end>n_endpoints) {local_idx_end=n_endpoints;}

    for (size_t idx = local_idx_start; idx < local_idx_end; idx++) {
        const endpoint& my_ep = endpoints[idx];
        if (my_ep.t == endpoint::SUBSCRIPTION) {
            if (my_ep.e == endpoint::LOWER) {
                // std::cout << "S I " << my_ep.id << std::endl;
                my_sub.insert(my_ep.id);
            } else {
                // std::cout << "S R " << my_ep.id << std::endl;
                my_sub.remove(my_ep.id);
                nmatches += my_upd.size();
            }
        } else {
            if (my_ep.e == endpoint::LOWER) {
                // std::cout << "I R " << my_ep.id << std::endl;
                my_upd.insert(my_ep.id);
            } else {
                // std::cout << "I R " << my_ep.id << std::endl;
                my_upd.remove(my_ep.id);
                nmatches += my_sub.size();
            }
        }
    }
*(d_nmatches+my_rank)=nmatches;
}
```

Infine, è stata realizzata la funzione main che si occupa delle chiamate delle funzioni sopra presentate e degli eventuali timer per verificare le prestazioni dell'esecuzione dell'algoritmo. Di questa parte, composta da istruzioni

di assegnazione e controllo dei parametri, si commenta solo l'utilizzo della libreria Thrust:

```
//vengono creati sulla memoria della gpu per contenere gli
//intervalli sub e upd, vengono inizializzati nel costruttore
thrust::device_vector<struct interval> sub(sub_in,sub_in+n);
thrust::device_vector<struct interval> upd(upd_in,upd_in+m);
```

e sulle chiamate dei vari step dell'algoritmo, chiaramente, con calcolo del tempo impiegato e relativo output su console:

```
//Inizializzo il tempo per calcolare il passo 1
timing_init( &timing );
timing_start( &timing );
//Passo 1 parallelo
cuda_pass_1<<<n_blocks,n_threads>>>(n_endpoints,  n,  m,
            d_my_sub_insert,
            d_my_sub_delete, d_my_upd_insert,
            d_my_upd_delete, d_endpoints);
checkCuda(cudaGetLastError());

//attende che il kernel precedente finisca
checkCuda(cudaDeviceSynchronize());

//Stampo il tempo per il passo 1
timing_stop( &timing );
total_time = timing_get_average( &timing );
std::cout << "pass 1 scan  time seconds " << total_time << std::endl;
fflush (stdout);
```

In tutti i passaggi, va sottolineato, si è realizzata l'esecuzione attendendo la terminazione di tutti i thread allocati e controllando, ad ogni assegnamento, che non vi fossero errori nell'architettura CUDA:

```
//Inizializzo il tempo per il passo 2
timing_init( &timing );
timing_start( &timing );
```

```
//Passo 2 seriale
cuda_pass_2<<<1,1>>>(d_my_sub_insert, d_my_sub_delete, d_my_upd_insert,
            d_my_upd_delete, d_subscriptions, d_updates, maxproc,n,m);
checkCuda(cudaGetLastError());

//attende che il kernel precedente finisca
checkCuda(cudaDeviceSynchronize());

//Stampa il tempo impiegato per il passo 2
timing_stop( &timing );
total_time = timing_get_average( &timing );
std::cout << "pass 2  merge / subtract time seconds " <<
                                      total_time << std::endl;
fflush (stdout);

//Inizializzo il tempo per il passo 3
timing_init( &timing );
timing_start( &timing );

//Passo 3 parallelo
cuda_pass_3<<<n_blocks, n_threads>>>(n_endpoints, d_subscriptions,
                                  d_updates, d_endpoints, d_nmatches);
checkCuda(cudaGetLastError());

//attende che il kernel precedente finisca
checkCuda(cudaDeviceSynchronize());

//Stampa il tempo impiegato per il passo 3 scan
timing_stop( &timing );
total_time = timing_get_average( &timing );
std::cout << "pass 3 scan time seconds " << total_time << std::endl;
fflush (stdout);
```

Si termina l'esecuzione del main richiamando «thrust::reduce» e ritornando la somma finale ottenuta.

```
size_t sum = thrust::reduce(nmatches.begin(),
                nmatches.end(), (size_t) 0, thrust::plus<size_t>());
```

```
/* i vettori vengono cancellati automaticamente quando la funzione
                                                    effettua il return */
return sum;
```

«thrust::reduce» consiste in una generalizzazione della sommatoria. Ovvero, svolge un'operazione di sommatoria (binaria) di tutti gli elementi nell'intervallo [first, last]). Questa operazione di reduce è simile ad std::accumulate della libreria C++.
La differenza principale tra le due funzioni è che std::accumulate garantisce l'ordine di sommatoria, mentre reduce richiede l'associatività dell'operazione binaria per parallelizzare la riduzione.
Dopo una prima analisi delle prestazioni, descritte dettagliatamente in seguito, si è notato che le stesse risultavano nettamente inferiori a quelle dell'algoritmo di partenza su CPU. Di conseguenza, si è pensato di parallelizzare ulteriormente alcuni step dell'algoritmo su GPU per migliorarle.
Anzitutto, si è optato per la creazione di una funzione CUDA per le operazioni di Merge e Substract precedentemente descritte:

```
__global__ void cuda_merge_parallel(unsigned int *data,
                unsigned int *data2, unsigned int buflen,
                unsigned int *count_array)
{
    // total number of threads in kernel
    int thread_count=blockDim.x*gridDim.x;
    int my_rank=blockIdx.x*blockDim.x + threadIdx.x;

    // divide and calculate start and end for this thread
    const int local_n = (buflen + thread_count - 1) / thread_count;
    const int local_idx_start = my_rank*local_n;
    int local_idx_end = local_idx_start + local_n;
    if (local_idx_end>buflen) {local_idx_end=buflen;}

count_array[my_rank]=0;
    for (size_t idx = local_idx_start; idx < local_idx_end; idx++) {
    data[idx]|=data2[idx];
        count_array[my_rank] += __popc(data[idx]);
}
```

```
}

__device__ void cuda_merge(unsigned int *data, unsigned int *data2,
                           unsigned int buflen, size_t *count)
{
    const int n_blocks=10;
    const int n_threads=1;
    unsigned int *count_array=new unsigned int[n_threads*n_blocks];
    *count=0;
    cuda_merge_parallel<<<n_blocks,n_threads>>>(data,data2,
                                                buflen,count_array);
    cudaDeviceSynchronize();
    for (int ii=0;ii<n_threads*n_blocks;++ii) {
        (*count)+=count_array[ii];
    }
    delete [] count_array;
}

__global__ void cuda_subtract_parallel(unsigned int *data,
                    unsigned int *data2, unsigned int buflen,
                    unsigned int *count_array)
{
    // total number of threads in kernel
    int thread_count=blockDim.x*gridDim.x;
    int my_rank=blockIdx.x*blockDim.x + threadIdx.x;

    // divide and calculate start and end for this thread
    const int local_n = (buflen + thread_count - 1) / thread_count;
    const int local_idx_start = my_rank*local_n;
    int local_idx_end = local_idx_start + local_n;
    if (local_idx_end>buflen) {local_idx_end=buflen;}

    count_array[my_rank]=0;
    for (size_t idx = local_idx_start; idx < local_idx_end; idx++) {
        data[idx] = data[idx] & ~data2[idx];
        count_array[my_rank] += __popc(data[idx]);
    }
```

```
}

__device__ void cuda_subtract(unsigned int *data,
                    unsigned int *data2, unsigned int buflen,
                    size_t *count)
{
    const int n_blocks=10;
    const int n_threads=1;
    unsigned int *count_array=new unsigned int[n_threads*n_blocks];
    *count=0;
    cuda_subtract_parallel<<<n_blocks,n_threads>>>(data,data2,
                                                   buflen,count_array);
    cudaDeviceSynchronize();
    for (int ii=0;ii<n_threads*n_blocks;++ii) {
        (*count)+=count_array[ii];
    }
    delete [] count_array;
}
```

Dopo un'attenta analisi del codice realizzato, si è pensato di parallelizzare anche le operazioni di copiatura e di salvataggio dei dati:

```
__global__ void cuda_memcpy_parallel(unsigned int *data,
                unsigned int *data2, unsigned int buflen)
{
    // total number of threads in kernel
    int thread_count=blockDim.x*gridDim.x;
    int my_rank=blockIdx.x*blockDim.x + threadIdx.x;

    // divide and calculate start and end for this thread
    const int local_n = (buflen + thread_count - 1) / thread_count;
    const int local_idx_start = my_rank*local_n;
    int local_idx_end = local_idx_start + local_n;
    if (local_idx_end>buflen) {local_idx_end=buflen;}

memcpy( data+local_idx_start, data2+local_idx_start,
                    (local_idx_end-local_idx_start)*sizeof(*data));
}
```

```
__device__ void cuda_memcpy(unsigned int *data, unsigned int *data2,
                            unsigned int buflen)
{
    const int n_blocks=1;
    const int n_threads=4;
    cuda_memcpy_parallel<<<n_blocks,n_threads>>>(data,data2,buflen);
    cudaDeviceSynchronize();
}

__global__ void cuda_memset_parallel(unsigned int *data,
                        unsigned int val, unsigned int buflen)
{
    //total number of threads in kernel
    int thread_count=blockDim.x*gridDim.x;
    int my_rank=blockIdx.x*blockDim.x + threadIdx.x;

    // divide and calculate start and end for this thread
    const int local_n = (buflen + thread_count - 1) / thread_count;
    const int local_idx_start = my_rank*local_n;
    int local_idx_end = local_idx_start + local_n;
    if (local_idx_end>buflen) {local_idx_end=buflen;}

    memset( data+local_idx_start, val,
                (local_idx_end-local_idx_start)*sizeof(*data));
}
```

Queste ultime funzioni sono state richiamate tramite «cuda_memset» e si è dovuto richiamare la «cudaDeviceSynchronize();» per attendere la corretta esecuzione delle operazioni. A seguito di queste modifiche, le prestazioni dell'algoritmo sono migliorate di 10 volte rispetto la prima versione.

4.5 Problemi Riscontrati

In sintesi, le difficoltà principali riscontrate durante le operazioni di adattamento e ottimizzazione dell'algoritmo, per l'applicazione in GPU-CUDA

sono derivate dall'assenza di criteri guida per la realizzazione dell'algoritmo. Inoltre, sono stati affrontati diversi problemi di riorganizzazione dei dati per adattare l'esecuzione dell'algoritmo in parallelo, così da permettere una verifica attendibile dell'affidabilità e della correttezza dei risultati ottenuti negli output. Di nuovo, si vuole notare che la libreria thrust ha permesso di semplificare il processo in modo determinante. Tuttavia, la parte restante di programmazione, ha richiesto numerose ore di lavoro visto e considerato che, primo, si approcciava questa nuova tecnologia per la prima volta, secondo, il materiale a disposizione per sviluppare delle soluzioni, di conseguenza, era poco, e terzo, le operazioni da svolgere a primo impatto erano tutto furochè banali.
Infine, si segnala che in termini di paragone, una delle difficoltà principali è derivata dagli hardware: dovendo basarsi su quelli a disposizione, in cui tutti presentavano una CPU qualitativamente migliore della GPU, capire quale dei due fosse il dispositivo di calcolo più efficiente è stato decisamente complicato.

Capitolo 5

Analisi delle Prestazioni

Nel seguente capitolo si svolge l'analisi delle prestazioni degli algoritmi testati. Nel primo paragrafo si svolge una breve introduzione circa i criteri di valutazione adottati nella valutazione delle performance; nel secondo si presentano i risultati ottenuti e nel terzo si svolge un commento preliminare degli stessi.

5.1 Criteri di Valutazione

L'obiettivo dell'applicazione era quello di sviluppare un algoritmo in grado di eguagliare e/o superare il parallel sort in termini di prestazioni; in altre parole, di sviluppare un'applicazione in grado di processare i dati in un tempo inferiore. In questo senso, un primo criterio fondamentale da considerare è la potenza dell'hardware installato nel computer utilizzato per calibrare l'algoritmo, specialmente in proporzione all'hardware a disposizione nei test condotti poi sul server dell'università.
Il confronto, da cui si sono tratte le valutazioni rispetto alle prestazioni dell'applicazione sviluppata, si è basato sui tempi di esecuzione dei tre algoritmi descritti nei capitoli precedenti: il brute force («brute», nei grafici e tabelle); l'applicazione GPU-CUDA basata sul sort-based parallel («CUDA») e il sort-based parallel («sort»). Si noti che il confronto si è basato sul solo parametro delle tempistiche per la mancanza, ad oggi, di standard e regole ufficiali atte a determinare l'efficacia e l'economicità di un algoritmo in questo ambito di applicazione.
Va sottolineato che i dati utilizzati sono fittizi, ossia generati dal program-

ma. In un'applicazione reale, con dati estrapolati da situazioni verosimili, gli output potrebbero variare in modo significativo. Inoltre, nella lettura di tabelle e grafici dei prossimi paragrafi, va tenuto a mente che i dati rappresentati sono stati ottenuti dopo diverse esecuzioni dei tre algoritmi; in altre parole, le cifre ed i grafici che rappresentano le prestazioni messe a confronto, si riferiscono alla media dei risultati, ossia delle run ripetute, così da ottenere una rappresentazione il più verosimile ed affidabile possibile.
Si tenga presente, infatti, che durante i test degli algoritmi, sia il proprio computer utilizzato (che per semplicità chiameremo «pc») sia il server dell'università (che per semplicità chiameremo «server») stavano verosimilmente processando anche altre applicazioni - ad esempio applicazioni in background, nel pc; altri algoritmi in fase di prova, nel server -. In altre parole, la loro potenza di calcolo non era dedicata al 100% all'esecuzione dei test in oggetto. Il corollario di queste precisazioni è che i dati, i grafici e le tabelle risultanti rappresentano prestazioni ottenute entro dette contingenze: se non ottimali, sono perlomeno contestualizzate e ciò permette di inferire delle conclusioni attendibili.

5.2 Risultati Ottenuti

In questo paragrafo si commentano i risultati ottenuti alla luce delle specifiche sopradette, facendo attenzione a dare una spiegazione del "comportamento" dei tre algoritmi messi a confronto che si inferisce dall'andamento dei grafici cartesiani e dalle tabelle numeriche riportate di seguito. Si noti, ai fini di una corretta lettura dei risultati e un loro confronto, che la prima versione dell'algoritmo realizzato adattando il sort-based è stata testata solamente nel «pc»; mentre la versione migliorata è stata testata sia sul «pc» sia sul «server».
Si ritiene doveroso un appunto, le specifiche del «server» utilizzato sono queste:

- CPU: 12 Core,
- GPU: «NVIDIA GTX 1070».

Quelle del «pc» le seguenti:

- CPU: 12 Core,

- GPU: «NVIDIA GTX 1650».

Specifiche che giustificano, specie guardando alla GPU, la differenza in termini di prestazioni: la GTX 1070 riporta un «Average 1080p FPS» di 101.1, «Average 1440p FPS», 74.7 e un «Average 4K FPS» di 44.6. La GTX 1650 invece, riporta un «Average 1080p FPS» di 73.2, «Average 1440p FPS» di 53.8 e un «Average 4K FPS» di 32.1.

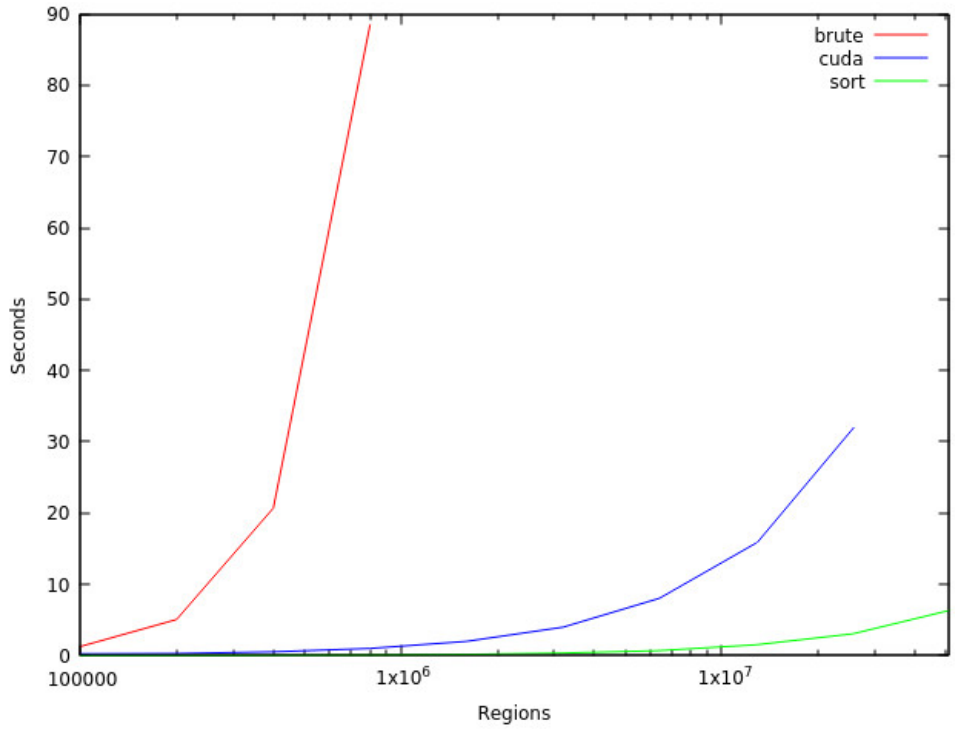

Figura 5.1: Rappresentazione delle prestazioni degli algoritmi testati nel pc nella prima fase. (elaborazione dell'autore).

Nel grafico riferito agli algoritmi nella prima fase operato nel «pc» (vedi Figura 5.1) si nota una lampante differenza prestazionale (ossia, si ricordi, di tempo impiegato in secondi, «seconds») tra il brute force - che anche con un lieve aumento delle region si abbassano drasticamente e tende a infinito - e quelle di CUDA e sort, che invece si mantengono perlopiù stabili e paragonabili, almeno fino a $1x10^7$ regions, seppure sort presenti prestazioni

sensibilmente superiori ben prima delle $1x10^6$ regions.
Nel secondo grafico (vedi Figura 5.2) vengono rappresentati i dati riferiti alle prestazioni degli algoritmi successivamente all'ottimizzazione dell'applicazione in GPU-CUDA.

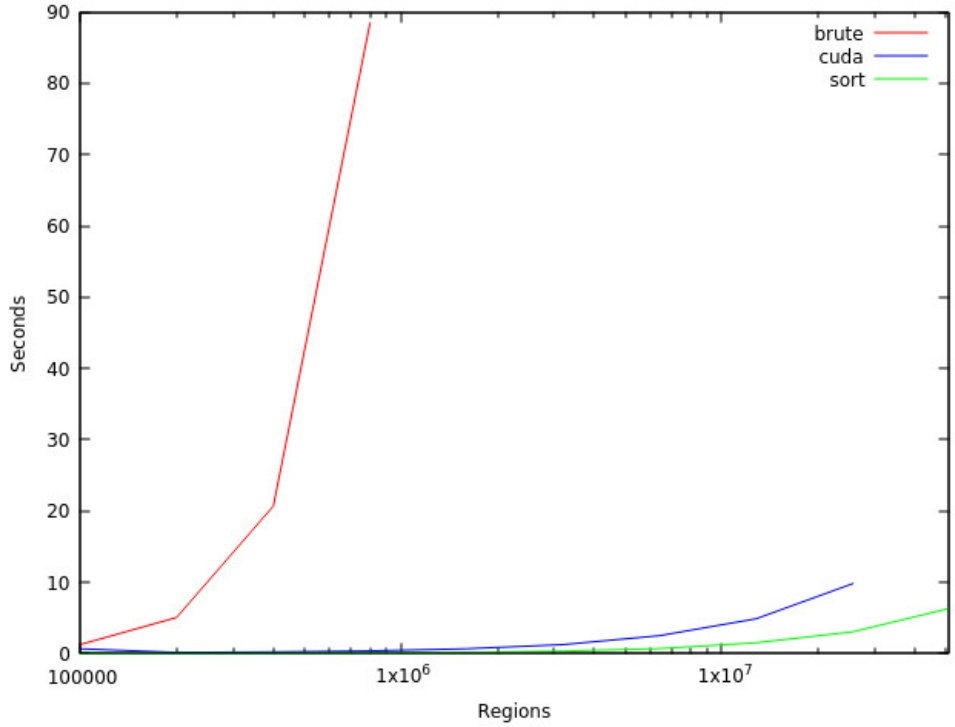

Figura 5.2: Rappresentazione delle prestazioni degli algoritmi testati nel pc dopo l'ottimizzazione dell'applicazione in GPU-CUDA. (elaborazione dell'autore)

Qui, tralasciando la spezzata rappresentante il brute-force che rimane la stessa, si nota immediatamente l'appiattimento della spezzata riferita a CUDA lungo l'asse delle ascisse, quasi a ricalcare quello dell'algoritmo sort: più in particolare, ciò che salta all'occhio sono le prestazioni riferite alla sezione che va da 100000 a $1x10^6$, dove le due spezzate appaiono sostanzialmente sovrapposte, grazie al miglioramento dell'algoritmo (citato nel precedente capitolo) implementato in GPU-CUDA pari a x10 rispetto la versione precedente. Da $1x10^6$, poi, di nuovo si nota una differenza sensibile, seppure non

determinante, fra sort e CUDA: le due spezzate tendono infatti a divaricarsi da $1x10^7$ in avanti; precisamente, a parità di region processate, la spezzata CUDA non è che una proiezione leggermente più "lenta" (in termini cartesiani, traslata verso destra e in basso) di quella sort.
Di seguito, per rendere più leggibile il confronto tra le prestazioni degli algoritmi presi in esame, nei grafici a seguire (vedi Figure 5.3 e 5.4) si riporta una variazione dei precedenti grafici in cui viene applicata un'operazione di logaritmo per facilitare la visualizzazione e la lettura dei risultati dei tempi delle prestazioni registrate durante i test.

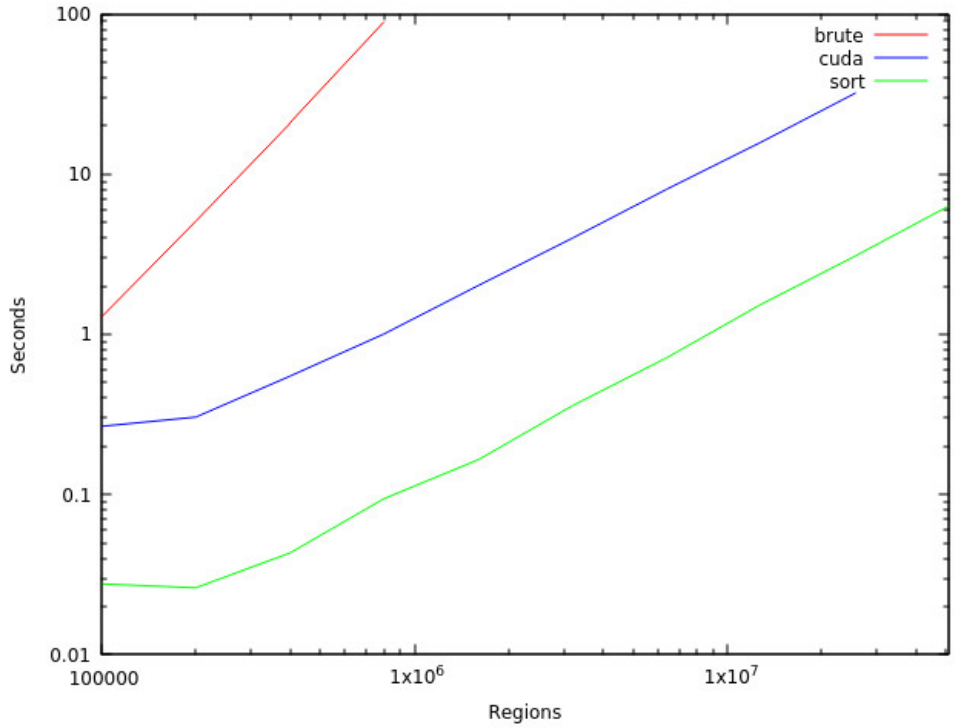

Figura 5.3: Rappresentazione delle prestazioni degli algoritmi testati nel pc nella prima fase con operazione di logaritmo sul risultato. (elaborazione dell'autore).

Successivamente, alla luce dei risultati ottenuti, si è pensato di verificare le prestazioni anche sul server e si sono estrapolati ulteriori grafici basati sui nuovi tempi registrati:

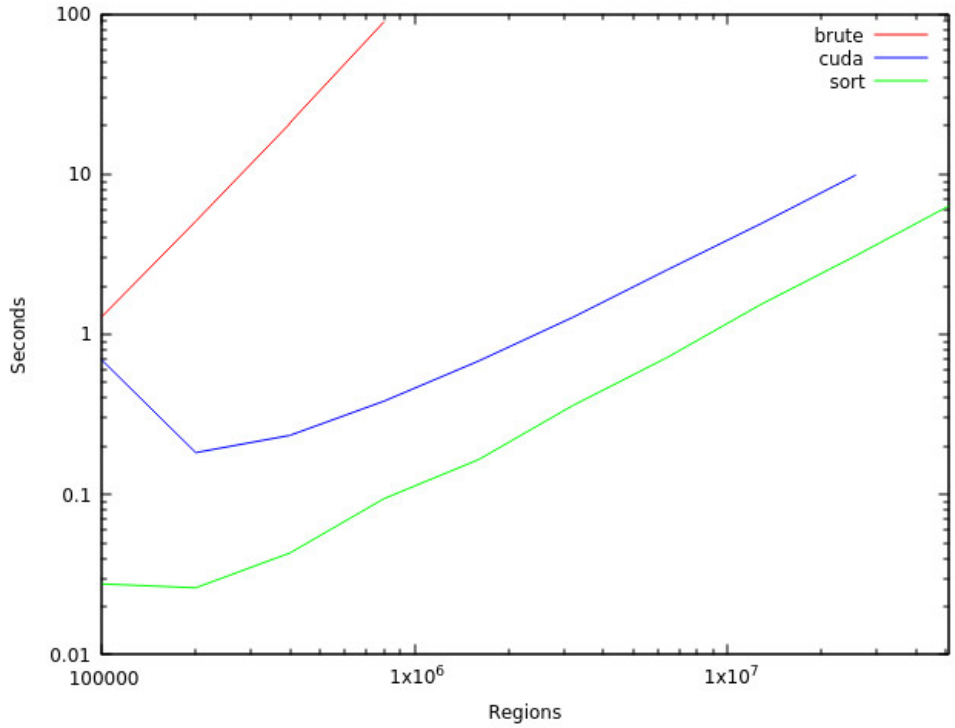

Figura 5.4: Rappresentazione delle prestazioni degli algoritmi testati nel pc dopo l'ottimizzazione dell'applicazione in GPU-CUDA con operazione di logaritmo sul risultato. (elaborazione dell'autore).

Il grafico (vedi Figura 5.5) si riferisce alle prestazioni dell'algoritmo ottimizzato, sul «server». Per semplicità di visione e comprensione dei risultati si è deciso di testare solamente la versione migliorata dell'algoritmo, dato che la prima versione risultava palesemente più lenta nell'esecuzione.
Di nuovo - trascurando la spezzata del brute-force, che rimane sostanzialmente la stessa -, si registra un appiattimento della spezzata del CUDA lungo l'asse delle ascisse che ricalca quella del sort, come visto nella prova sul proprio computer. In tal senso, la continuità fra i due test offre una prova dell'affidabilità della valutazione, seppur entro le contingenze descritte nel primo paragrafo.
Anche in questo caso, le spezzate di CUDA e sort da 100000 a $1x10^6$ sono sovrapposte, mentre da $1x10^6$, la differenza si fa sensibile, con una marcata divaricazione a partire da $1x10^7$ in avanti, dando di nuovo la sensazione

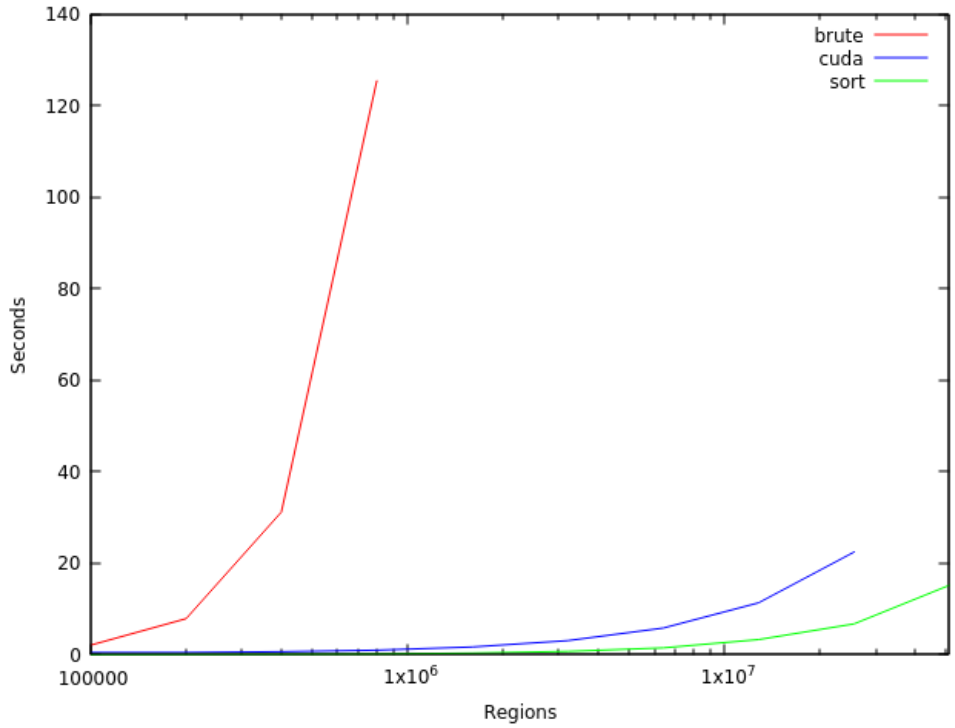

Figura 5.5: Rappresentazione delle prestazioni degli algoritmi testati nel server dopo l'ottimizzazione dell'applicazione in GPU-CUDA. (elaborazione dell'autore).

che, a parità di region processate, la spezzata CUDA sia una proiezione leggermente traslata del sort. Ma è il grafico che riporta i risultati con l'applicazione del logaritmo a rivelare la sostanziale differenza nei risultati ottenuti dal CUDA (vedi Figura 5.6).

Si nota infatti che la curva dell'algoritmo CUDA ottimizzato, seppure leggermente, non è parallela a quella del sort: per un certo numero $1x10n$ region, dunque, le prestazioni del primo eguagliano e superano quelle del secondo.
Infine, si noti, le prestazioni registrate sul «server» sono leggermente inferiori rispetto a quelle del test sul «pc». Di conseguenza, anche le prestazioni sul «pc», dopo un certo numero di region, eguaglieranno e supereranno le prestazioni del sort-based.

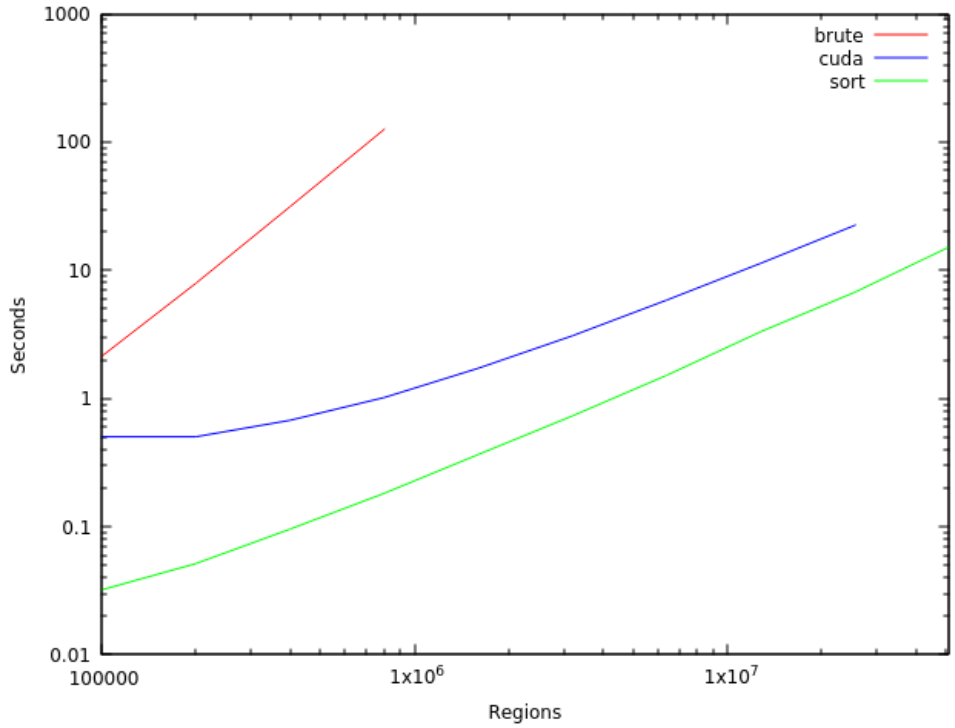

Figura 5.6: Rappresentazione delle prestazioni degli algoritmi testati nel server dopo l'ottimizzazione dell'applicazione in GPU-CUDA con operazione di logaritmo sul risultato. (elaborazione dell'autore).

5.3 Tabelle Risultanti

Nelle seguenti tabelle, si è deciso di riportare alcuni dati sostanziali ottenuti durante i vari test degli algoritmi sul proprio computer corredati da una breve descrizione dei risultati ottenuti:

Numero Region	Secondi
100.000	1.281814
200.000	5.079493
400.000	20.664122
800.000	88.407161

La tabella sopra citata riporta il tempo di esecuzione in secondi ed il numero di region.
Di seguito, un piccolo estratto del file di log dell'esecuzione dell'algoritmo brute-force:

```
# ./ddm compiled with:
#  CFLAGS=-Wall -std=c99 -pedantic -D_XOPEN_SOURCE=700
# -O3 -Wunknown-pragmas -fopenmp -DNDEBUG
#  CXXFLAGS=-Wall -std=c++17 -pedantic -DEPSET_RAW -O3
# -Wunknown-pragmas -fopenmp -DNDEBUG
#  LDFLAGS=-lm -lrt
#  LDLIBS=-lgomp
# OPENMP enabled with max 12 threads
# Algorithm ........... bfm
# Number of dimensions. 1
# alpha................ 0.010000
# RNG seed ............ 1234
# Number of regions.... 800000
3837 matches 88.407161
```

Per quanto riguarda l'esecuzione del sort-based, si sono ottenuti i seguenti risultati:

Numero Region	Secondi
100.000	0.027709
200.000	0.026190
400.000	0.043138
800.000	0.094429
1.600.000	0.165081
3.200.000	0.358552
6.400.000	0.714177
12.800.000	1.535982
25.600.000	3.075795
51.200.000	6.348201

```
# ./ddm compiled with:
#  CFLAGS=-Wall -std=c99 -pedantic -D_XOPEN_SOURCE=700
```

```
# -O3 -Wunknown-pragmas -fopenmp -DNDEBUG
#  CXXFLAGS=-Wall -std=c++17 -pedantic -DEPSET_RAW -O3
# -Wunknown-pragmas -fopenmp -DNDEBUG
#  LDFLAGS=-lm -lrt
#  LDLIBS=-lgomp
# OPENMP enabled with max 12 threads
# Algorithm ........... sbm
# Number of dimensions. 1
# alpha................ 0.010000
# RNG seed ............ 1234
# Number of regions.... 51200000
Timer[Parallel SBM] 6.34815
304501 matches 6.348201
```

Infine, i dati riguardanti l'esecuzione dell'algoritmo realizzato (si sono presi in esame i dati della prima versione e della seconda versione ottimizzata):

Numero Region	Secondi
100.000	0.266323
200.000	0.304071
400.000	0.547180
800.000	1.005952
1.600.000	2.016379
3.200.000	3.983100
6.400.000	8.047253
12.800.000	15.809675
25.600.000	31.929907

```
# ./ddm compiled with:
#  CFLAGS=-Wall -std=c99 -pedantic -D_XOPEN_SOURCE=700
# -O3 -Wunknown-pragmas -fopenmp -DNDEBUG
#  CXXFLAGS=-Wall -std=c++17 -pedantic -DEPSET_RAW -O3
# -Wunknown-pragmas -fopenmp -DNDEBUG
#  LDFLAGS=-lm -lrt
#  LDLIBS=-lgomp
# OPENMP enabled with max 12 threads
```

```
# Algorithm ..... cuda
# Number of dimensions. 1
# alpha.............. 0.010000
# RNG seed ..... 1234
# Number of regions.... 25600000
using cuda threads 10
copy time seconds 1.64733
sort time seconds 0.541283
init data seconds 0.00112564
pass 1 scan  time seconds 3.29331
pass 2  merge / subtract time seconds 2.89688
pass 3 scan time seconds 2.73985
76388 matches 11.121946
```

Per la versione finale ed ottimizzata risultano i seguenti tempi di esecuzione:

Numero Region	Secondi
100.000	0.693773
200.000	0.182790
400.000	0.233898
800.000	0.382563
1.600.000	0.680137
3.200.000	1.274990
6.400.000	2.516560
12.800.000	4.902132
25.600.000	9.815196

Successivamente, si è testata l'esecuzione degli algoritmi sul server dell'università, di seguito si riportano le tabelle ottenute (nell'ordine: brute-force, sort-based ed applicativo GPU-CUDA):

Numero Region	Secondi
100.000	2.129678
200.000	7.895433
400.000	31.225321
800.000	125.282111

Numero Region	Secondi
100.000	0.032063
200.000	0.051403
400.000	0.095402
800.000	0.181687
1.600.000	0.365756
3.200.000	0.730854
6.400.000	1.515949
12.800.000	3.321081
25.600.000	6.753501
51.200.000	15.186078

Numero Region	Secondi
100.000	0.501622
200.000	0.501632
400.000	0.675211
800.000	1.013874
1.600.000	1.720523
3.200.000	3.076930
6.400.000	5.848090
12.800.000	11.342580
25.600.000	22.425330

5.4 Commenti

Le considerazioni che si possono svolgere rispetto ai grafici e alle tabelle riportanti i dati dei test sono sostanzialmente tre, tra loro strettamente correlate.
La prima è apparentemente tautologica: la potenza della GPU, lo si è notato nella differenza dall'entità delle prestazioni fra «pc» e «server», rimane il fattore prestazionale determinante, specie dal punto di vista economico; dunque, la soluzione più efficiente ed efficace è ottimizzare il codice.
Alla luce di questa, però, se si riconsiderano più attentamente i risultati ottenuti si noterà che seppur la differenza a livello di prestazioni è sensibile, la progressione delle spezzate è e rimane pressoché la stessa.

Un'ipotesi preliminare è che l'hardware a disposizione abbia influito sulle prestazioni, specie la RAM della GPU a disposizione. Considerato ciò, si suppone che avendo a disposizione un hardware migliore sul computer personale e sul server - ossia delle GPU più potenti -, le prestazioni ottenute per il CUDA potrebbero avvicinarsi ulteriormente a quelle registrate per il sort, fino a raggiungere il punto in cui il CUDA le eguaglia e le supera.

Capitolo 6

Conclusioni e Sviluppi Futuri

In questa sede, si propongono alcune soluzioni per possibili sviluppi futuri; in particolare, si svolge una trattazione delle migliorie che si potrebbero implementare nell'algoritmo per ottimizzarne ulteriormente le prestazioni. Infine, si procede a svolgere le conclusioni rispetto all'oggetto della tesi.

6.1 Sviluppi Futuri

Alla luce di quanto rilevato nel capitolo precedente, dopo un'attenta analisi dei risultati e delle discussioni, si sono evinti diversi aspetti su cui intervenire per ottimizzare le prestazioni dell'algoritmo:

1. In primis, si potrebbe impiegare un dataset realmente esistente nel mondo reale, ossia, cercare di analizzare il problema che si sta cercando di risolvere concretamente. Attualmente, invece, dal codice esistente vengono generati solamente dataset casuali;

2. In secondo luogo, sarebbe utile eseguire la computazione dell'algoritmo in una GPU con più RAM;

3. In terzo luogo, si dovrebbe eseguire una profilazione e un'ottimizzazione localizzata delle parti di codice adattate in questo elaborato e non riscritte;

4. Infine, si potrebbero sviluppare implementazioni alternative riscrivendo il codice da zero, ripartendo dal punto di partenza e pensando a nuove soluzioni da zero.

6.2 Conclusioni

L'obiettivo di questa tesi era di elaborare, descrivere e analizzare un algoritmo che risolvesse un problema di sort-matching all'interno di una simulazione, utilizzando una GPU su architettura CUDA.
In questa tesi, coerentemente, sono state progettate, implementate e valutate nuove implementazioni dell'algoritmo parallelo sort-based adattato alla GPU, sfruttando l'architettura CUDA a disposizione. Si è, quindi, cercato di raggiungere e superare le prestazioni su CPU dell'algortimo sort-based parallelo, nella soluzione del problema di matching e distribuzione dei dati all'interno del Data Distribution Management dello standard High-Level Architecture per simulazioni distribuite e parallele.
Nello specifico, si è iniziato analizzando attentamente il problema posto e cercando di comprendere e scorporare ogni parte dell'algoritmo sort-based. Successivamente, si è modificato l'algoritmo così da adattarlo all'esecuzione su GPU-CUDA. Dunque, alla luce dei dati e delle valutazioni eseguite nella prima versione, si è proceduto con ottimizzazioni ulteriori. Va notato, poi, che si è optato per l'assegnazione di ulteriore memoria ad ogni thread dell'algoritmo e alla parallelizzazione delle operazioni seriali che impiegavano più tempo di esecuzione.
In altre parole: si è cercato di parallelizzare l'operazione di «merge» e copiatura dei dati; queste modifiche, a loro volta, hanno migliorato le prestazioni dell'algoritmo di 10 volte rispetto alla versione precedente, avvicinandosi molto alle performance dell'algoritmo sort-based su CPU.
In conclusione, si vuole sottolineare che, durante l'implementazione e i test, si è accusata la mancanza di standard e materiali da seguire per la valutazione dell'algoritmo, specie rispetto alla sua correttezza.
Molti studi si focalizzano su algoritmi che generano sovra-insiemi della soluzione ottima; allo stesso modo, si è cercato di realizzare un algoritmo che evitasse trasferimenti di dati non necessari tra i federati, così da limitare il più possibile la generazione di falsi positivi.
Di fatto, la generazione di un sovra-insieme non ha costo necessariamente quadratico. Questo tipo di approccio può essere decisamente vantaggioso in caso la rete di interconnessione tra i federati sia sovradimensionata. Dopotutto, il trasferimento di dati, non necessariamente incrementa il "peso" sul collo di bottiglia del sistema. Avvicinandosi al limite di portata della rete utilizzata le performance calano drasticamente; il sistema, a questo punto,

si congestiona velocemente.
Infine, una perplessità: la maggior parte degli algoritmi per il DDM vengono presentati in dettaglio in articoli e pubblicazioni scientifiche, ma le effettive implementazioni non sono disponibili (nonostante raramente abbiano fini commerciali), rendendo necessario, per chiunque voglia effettuare un confronto, un dispendio di tempo e fatica per implementare nuovamente l'algoritmo altrui, rischiando, comè ovvio, l'implementazione risulti sbagliata, falsando i confronti. Si spera, in questo senso, che la consuetudine cambi, così da permettere una ricerca e sviluppo nel campo più spedita, aperta e trasparente. Anche per questo, si ringraziano sentitamente gli autori dell'articolo da cui è stato tratto il codice per lo studio e la programmazione delle implementazioni trattate, nonché gli algoritmi utilizzati per operare il confronto dei risultati. [19] Per quanto riguarda chi scrive, si noti, il codice sviluppato in questa tesi verrà reso disponibile in licenza open source. [22]

Bibliografia

[1] Kuhl F et al., (1999).
Creating Computer Simulation Systems: An Introduction to the High Level Architecture.
New York: Prentice Hal.

[2] Morse, K.L., & Petty, M.D. (2004).
High Level Architecture Data Distribution Management migration from DoD 1.3 to IEEE 1516.
Concurrency and Computation: Practice and Experience, 16.

[3] Boukerche, A. and Gu, Y. (2013).
Data Distribution Management. In Large Scale Network-Centric Distributed Systems.
(eds H. Sarbazi-Azad and A.Y. Zomaya).

[4] Morse K. L., Steinman J. S. (1997).
Data Distribution Management in the HLA: Multidimensional Regions and Physically Correct Filtering.
In: Proceedings of the 1997 Spring Simulation Interoperability Workshop, 343 - 352.

[5] Tacic I., Fujimoto R (1997).
Synchronized Data Distribution Management in Distributed Simulation.
In: Proceedings of the 1997 Spring Simulation Interoperability Workshop.

[6] Boukerche A., Roy A. (2000).
In Search of Data Distribution Management in Large Scale Distributed Simulations.
In: Summer Computer Simulation Conference (SCSC). The Society

for Modeling and Simulation International (SCS), IEEE Conference Publications.

[7] Raczy C., Tan G., Yu J. (2005)
A Sort-Based DDM Matching Algorithm for HLA.
In: ACM Transactions on Modeling and Computer Simulation (TOMACS), 14 - 38. ACM.

[8] Fujimoto R., McLean T., Perumalla K., Tacic I. (2000).
Design of High Performance RTI Software.
In Proceedings of Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS - RT 2000), 89 - 96. IEEE Conference Publications.

[9] Ayani R., Moradi F., Tan G. (2000).
Optimizing Cell-size in Grid-Based DDM.
In Proceedings of Fourteenth Workshop on Parallel and Distributed Simulation (PADS 2000), 93 - 100. IEEE Conference Publications.

[10] Petty M. D., Morse K. L. (2000).
Computational Complexity of HLA Data Distribution Management.
In Proceedings of the 2000 Fall Simulation Interoperability Workshop, 2000.

[11] Van Hook D. J., Rak S. J., Calvin J. O. (1994).
Approaches to Relevance Filtering.
In Proceedings of the 11th Workshop on Standards for the Interoperability of Distributed Simulations, 367 - 369.

[12] Tan G., Ayani R., Zhang Y., Moradi F. (2000).
An Experimental Platform for Data Management in Distributed Simulation.
In: Proceedings of Simulation Technology and Training Conference, 2000, 371 - 376.

[13] Tan G., Ayani R., Zhang Y., Moradi. F. (2000).
Grid-Based Data Management in Distributed Simulation.
In: Proceedings of the 33rd Annual Simulation Symposium, 20007 - 13.

[14] D. R. Musser.
Introspective sorting and selection algorithms.
Software Practice and Experience, 27(8): 983 - 993, 1997.

[15] Lo, S., Chung, Y., & Pai, F. (2010).
Offloading Region Matching of Data Distribution Management with CUDA.
2010 International Conference on Intelligent Systems, Modelling and Simulation, 306 - 311.

[16] Wang, Z., Liu, Y., & Chiu, S.C. (2014).
An efficient parallel collaborative filtering algorithm on multi-GPU platform.
The Journal of Supercomputing, 72, 2080 - 2094.

[17] Mangharam, R., & Saba, A.A. (2011).
Anytime Algorithms for GPU Architectures.
2011 IEEE 32nd Real-Time Systems Symposium, 47 - 56.

[18] Roberge, V., Tarbouchi, M., & Labonte, G. (2018).
Fast Genetic Algorithm Path Planner for Fixed-Wing Military UAV Using GPU.
IEEE Transactions on Aerospace and Electronic Systems, 54, 2105 - 2117.

[19] Marzolla, M., & D'angelo, G. (2020).
Parallel Data Distribution Management on Shared-memory Multiprocessors.
ACM Transactions on Modeling and Computer Simulation (TOMACS), 30, 1 - 25.

[20] Harris, Mark, Shubhabrata Sengupta, and John D. Owens.
"Parallel prefix sum (scan) with CUDA."
GPU gems 3.39 (2007): 851 - 876.

[21] Harris, Mark.
"Optimizing parallel reduction in CUDA."
Nvidia developer technology 2.4 (2007): 70.

[22] Poggi (2021).
https://github.com/GiovanniPoggi/DDMThesis.