ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Ingegneria e Architettura
Corso di Laurea in Ingegneria e Scienze Informatiche

Data Distribution Management: un approccio CUDA

Tesi di laurea in
High Performance Computing

Relatore
Prof. Moreno Marzolla

Presentata da
Alex Baiardi

III Sessione di Laurea
Anno Accademico 2020-2021

Indice

Elenco delle figure

Elenco degli algoritmi

Introduzione

In questa tesi affronteremo il problema del matching nel **Data Distribution Management**, ovvero la ricerca di intersezioni tra iper-rettangoli. Il Data Distribution Management è un servizio utilizzato nelle simulazioni parallele e distribuite. L'obiettivo è quello di trovare una possibile implementazione dell'algoritmo Sort Base Matching per GPU. In particolare, questo algoritmo risulta efficiente nella versione seriale, ma non è semplice da parallelizzare; infatti, non si conoscono soluzioni eseguite su GPU utilizzando CUDA.

In questo caso cercheremo di esplorare se l'obiettivo è raggiungibile e per farlo svilupperemo una versione semplificata, con il solo obiettivo che sia corretta. In questo primo momento teniamo in considerazione l'efficienza solo in modo marginale.

Nel primo capitolo di questa tesi si introduce il calcolo parallelo facendo riferimento a ciò che ha portato al suo sviluppo e quali sono le difficoltà che lo accompagnano. Inoltre, si descriverà il funzionamento delle GPU CUDA, che rappresentano una modalità di calcolo parallelo sempre più diffusa. Nel secondo capitolo vedremo il contesto in cui si inserisce questa tesi: viene definito cos'è il Data Distribution Management, qual è il problema che deve risolvere e i principali algoritmi che si conoscono. In seguito, vedremo la progettazione di un algoritmo parallelo adatto all'implementazione in CUDA che affronta una versione semplificata del problema. Infine, vedremo come viene implementato questo algoritmo e i limiti che presenta.

Capitolo 1

Calcolo parallelo

I calcolatori hanno assunto un ruolo molto importante nelle nostre vite; infatti, ci affidiamo sempre di più ad essi e vengono create ogni giorno applicazioni nuove. Certamente questo processo di digitalizzazione non è destinato a fermarsi, anzi aumenterà sempre più, tant'è che parliamo di transizione digitale. Quindi, avremo sempre più dati da elaborare e applicazioni sempre più varie e complesse. Tuttavia, spesso non ci rendiamo conto di quanti calcoli siano necessari per fare in modo che ciò sia possibile, probabilmente perché siamo abituati al fatto che i processori nel tempo hanno avuto prestazioni sempre migliori. Infatti, come analizzeremo nella sezione 1.1, abbiamo potuto contare su un aumento esponenziale delle prestazioni: come aveva intuito G. E. Moore. Però oggi abbiamo raggiunto dei limiti per cui la potenza di calcolo di un processore faticherà ad aumentare, ma non tutto è perduto; infatti, negli ultimi anni il miglioramento delle prestazioni sta passando e continuerà a passare attraverso il calcolo parallelo.

1.1 La legge di Moore

La Legge di Moore è una osservazione empirica formulata da Gordon Earle Moore nel 1965 [19]. Osservando gli sviluppi tecnologici che si stavano verificando, Moore affermò che il numero di transistor in un circuito integrato tende a raddoppiare ogni 24 mesi; da questo è derivato un corollario secondo cui anche le prestazioni dei calcolatori avrebbero seguito lo stesso andamento. Come si può vedere nel grafico nella figura 1.1, queste previsioni sono state confermate dalla storia: infatti, tra la metà degli anni '80 e i primi anni 2000 abbiamo assistito ad una crescita esponenziale delle prestazioni, per poi avere una attenuazione sempre maggiore. Questo rallentamento è dovuto al fatto che ci stiamo scontrando con dei limiti fisici. In particolare, le alte prestazioni necessitano di maggiore quantità di energia, la

quale produce eccessivo calore al punto da non riuscire a dissiparlo correttamente e quindi rendere i processori inaffidabili.

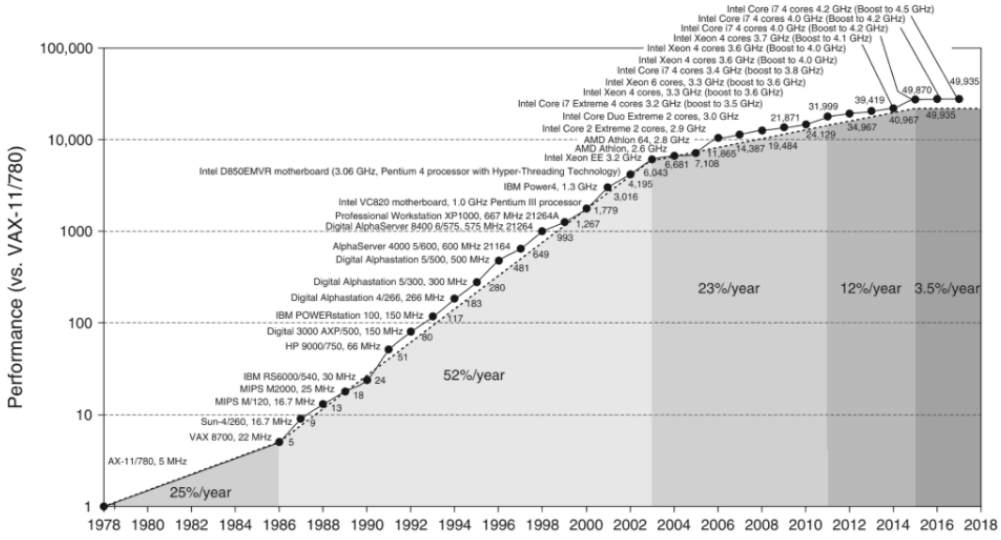

Figura 1.1: Aumento delle prestazioni dei processori nel tempo.
Fonte: `https://books.google.co.uk/books?id=cM8mDwAAQBAJ&pg=PA3`

L'aumento così rapido delle prestazioni ha fatto sì che vi fosse l'idea che qualora una applicazione avesse richiesto una maggiore potenza di calcolo, sarebbe stato sufficiente aspettare l'arrivo di processori più potenti. Tuttavia, ultimamente, visto che non si riesce più ad aumentare significativamente le prestazioni di un processore, invece che raddoppiarne la potenza, si raddoppia il numero di unità di calcolo: formando architetture parallele.

1.2 Potenzialità e problemi del calcolo parallelo

Il calcolo parallelo, ad oggi, è fondamentale in numerose applicazioni. Ad esempio, per sistemi in cui si elaborano grandi quantità di informazione sfruttando modelli matematici: previsioni meteo, previsioni sui cambiamenti climatici, simulazioni fisiche e modellazioni astronomiche.

L'idea che porta a sfruttare il calcolo parallelo è che, se ci vuole una potenza di calcolo n volte maggiore a quella di un processore, sarà sufficiente utilizzare n processori; nella pratica tuttavia ciò è molto raro e complesso. Infatti, questo implica che gli algoritmi siano perfettamente parallelizzabili e che tutte le unità di lavoro impieghino lo stesso tempo di esecuzione. Questo raramente è possibile, infatti,

spesso gli algoritmi presentano alcune parti, più o meno estese, intrinsecamente seriali. Inoltre, per la gestione del parallelismo, vengono aggiunte istruzioni non necessarie per l'algoritmo in sé.
Un altro fattore limitante è che spesso non si ha la possibilità di sfruttare tutto l'hardware a disposizione esclusivamente per una applicazione: infatti lavoriamo sempre più in sistemi multitasking.

Un altro limite per le prestazioni di applicazioni parallele, e non solo, è la differenza tra la velocità a cui operano le memorie e quella dei processori: infatti, vi è un forte divario dovuto al fatto che la velocità dei processori è aumentata molto più rapidamente di quella delle memorie. Questa differenza porta a dover attuare opportune strategie in fase di programmazione, per cercare di ridurre la presenza di colli di bottiglia che influirebbero pesantemente sulle prestazioni.

Nonostante le criticità rilevate, la programmazione parallela anche se non in modo ottimo, può permettere di aumentare significativamente le prestazioni e sfruttare al massimo le architetture hardware che abbiamo a disposizione. Infatti, le architetture parallele sono presenti in numerosi dispositivi di uso quotidiano, ma spesso non sono sfruttate al massimo perché la gran parte del software che utilizziamo è seriale.

La sfida, oggi, sta nel riuscire a rendere il software parallelo diffuso allo stesso modo in cui esistono architetture parallele. Tuttavia, il problema principale nella diffusione di software parallelo è che non esistono dei meccanismi automatici per trasformare il software seriale in software parallelo, ma è necessario che sia programmato in modo esplicito. Inoltre, esistono diversi tipi di architetture parallele ognuna con peculiarità e necessità specifiche: ad esempio l'architettura CUDA che viene descritta nella sezione 1.4. Perciò per fare in modo che il software parallelo si diffonda, è necessario avere a disposizione figure professionali specializzate in questo campo e al giorno d'oggi queste figure non sono molto diffuse; quindi, è fondamentale formare i programmatori affinché sappiano scrivere codice parallelo.

1.3 GPGPU

L'idea di sfruttare tutte le risorse hardware a disposizione, ha fatto in modo che si pensasse di utilizzare le GPU (Graphics Processing Units), non solo per la gestione della grafica, quale era il loro scopo originariamente e in cui erano specializzate, ma sfruttare il parallelismo massivo offerto dalle loro architetture per applicazioni generiche, dando vita al concetto di GPGPU (General-Purpose GPU). Nella figura 1.2, possiamo vedere schematicamente la differenza tra l'architettura di una CPU multicore e quella di una GPU, in particolare si nota l'elevato numero di core presenti in una GPU.

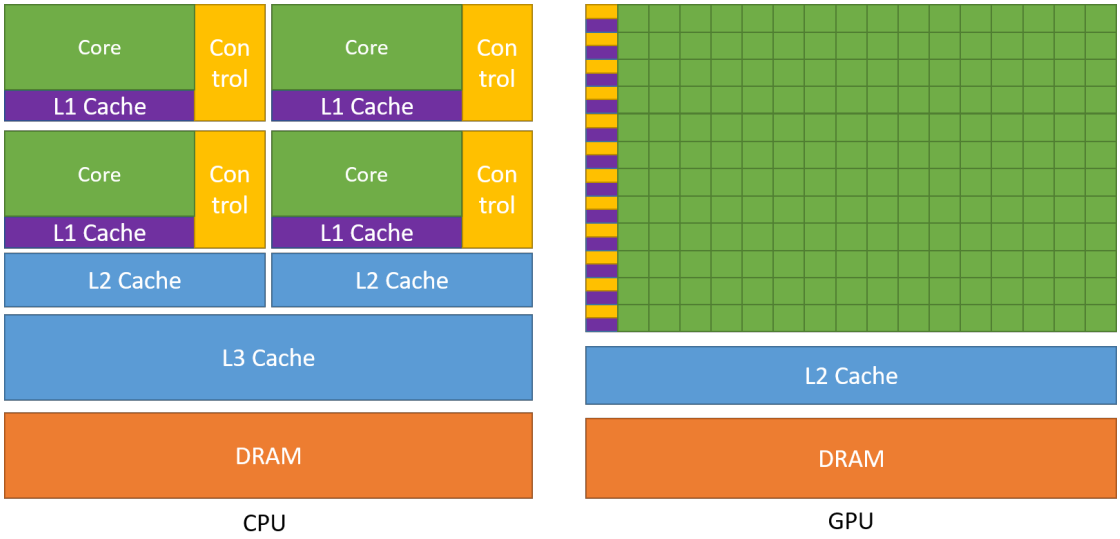

Figura 1.2: A sinistra l'architettura di una CPU, a destra quella di una GPU. Fonte: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

La tendenza di usare le GPU per usi generali ha creato la necessità di avere un modo per programmarle; questo ha fatto sì che i costruttori sviluppassero appositi ambienti di sviluppo, ad esempio CUDA e OpenCL.

1.4 CUDA

L'acronimo CUDA (Compute Unified Device Architecture) indica un'architettura hardware per l'elaborazione parallela sfruttando GPU. La prima versione è stata rilasciata nel 2007 da NVIDIA che ne risulta tuttora proprietaria. Attraverso CUDA è possibile scrivere programmi che vengono eseguiti su una o più schede grafiche NVIDIA.

CUDA prevede una divisione tra CPU e GPU, chiamate rispettivamente *host* e *device*, le quali sono considerate come unità indipendenti, che cooperano in modo asincrono. In particolare, all'occorrenza la CPU sollecita la GPU affinché esegua determinate operazioni e mentre la GPU esegue la computazione, la CPU ha ripreso il controllo ed è libera di eseguire altro. L'host e il device lavorano su due memorie distinte, in particolare vedremo meglio in seguito come è organizzata quella del device. L'host ha il compito di preparare l'esecuzione del device allocando la memoria e copiandone i dati tramite opportune funzioni di libreria, per poi lanciare l'esecuzione e al termine, dopo essersi sincronizzati attraverso apposite funzioni, recuperare dalla memoria del device i risultati. Il codice eseguito sul

device viene appositamente compilato e viene eseguito partendo da particolari funzioni chiamate *kernel*; queste funzioni richiedo due parametri obbligatori, griglia e blocco, che descrivono come organizzare logicamente l'esecuzione.

1.4.1 Struttura logica delle unità di esecuzione

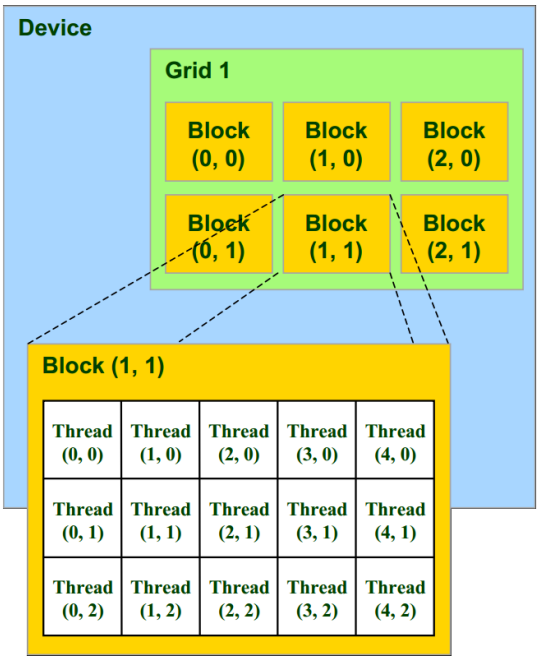

Figura 1.3: Esempio struttura logica delle unità di esecuzione CUDA.
Fonte: `https://ipython-books.github.io/58-writing-massively-parallel-code-for-nvidia-graphics-cards-gpus-with-cuda/`

CUDA definisce una struttura logica delle unità di esecuzione, in particolare:

- un **thread** indica la minima unità di esecuzione, tutti i thread di una griglia eseguono la stessa funzione kernel
- un **blocco** rappresenta una porzione indipendente di lavoro che può essere eseguita in qualunque ordine all'interno di una griglia. Viene definito come un array tridimensionale di thread, esiste un limite al numero massimo di thread presenti in un blocco, che dipende dalla versione di CUDA
- una **griglia** rappresenta un compito affidato alla GPU; è definita come un array di blocchi, bidimensionale nelle prime versioni, ora tridimensionale

I thread e i blocchi sono opportunamente indicizzati, con indici a 1, 2, o 3 dimensioni in base all'occorrenza. Nella figura 1.3 è possibile vederne una rappresentazione grafica.

Quando si scrive un kernel, non c'è la necessità di preoccuparsi di come questa struttura venga implementata sull'hardware, è sufficiente infatti tenere in considerazione alcuni elementi. In particolare, CUDA prevede di effettuare uno scheduling automatico al momento dell'esecuzione sfruttando al massimo l'hardware e risultandone indipendente. Sappiamo però che i thread vengono messi in esecuzione in gruppi da 32, chiamati *warp*. Ogni warp esegue le istruzioni simultaneamente, seguendo il paradigma SIMD (Single Instruction Multiple Data). Perciò, per ottenere la massima efficienza è bene che tutti i thread eseguano la stessa istruzione su dati diversi, questo però non costituisce un vincolo; infatti, ogni thread è libero di eseguire istruzioni diverse dagli altri: viene garantita una astrazione MIMD (Multiple Instruction Multiple Data). Tuttavia, questo comporta che nel caso vi sia una divergenza nel flusso di controllo, vengano eseguiti in modo seriale tutti i flussi, tenendo sospeso chi non deve eseguirli. Un esempio è il costrutto `if-else`, in cui se almeno un thread nel warp deve eseguire il ramo del vero e almeno uno il ramo del falso, verranno eseguiti entrambi, come è rappresentato nella figura 1.4.

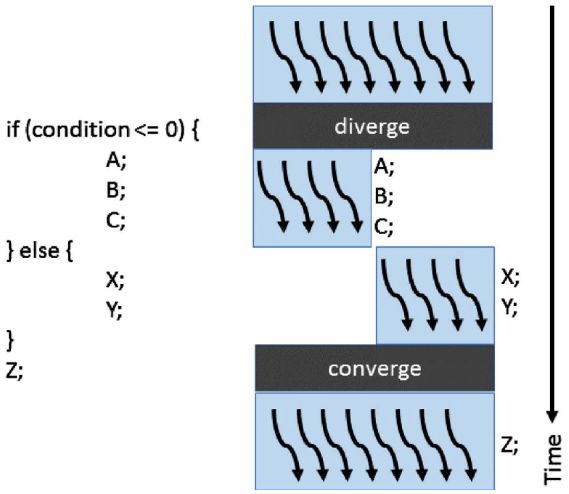

Figura 1.4: Esempio di divergenza tra i flussi di esecuzione di un warp.
Fonte: `https://www.researchgate.net/figure/Warp-divergence-due-to-if-else-statements_fig15_317608134/`

1.4.2 Struttura della memoria

Nella figura 1.5 è possibile vedere come è strutturata la memoria in un device CUDA, in particolare vediamo di seguito le caratteristiche di ogni tipologia:

- **memoria globale**: è la memoria più grande (alcuni GB), mantiene il suo stato per tutta la durata di esecuzione di una applicazione, è accessibile da tutti i thread della griglia, presenta una elevata latenza di accesso
- **memoria texture**: è una memoria di sola lettura ottimizzata per contenere dati di texture (alcuni MB), accessibile da tutti i thread della griglia
- **memoria delle costanti**: è una memoria persistente di sola lettura con bassa latenza, accessibile a tutti i thread della griglia, in genere di 64 KB
- **memoria condivisa**: è una memoria di circa 48 KB per blocco, accessibile da tutti i thread di un blocco, con una bassa latenza, non mantiene lo stato tra due kernel
- **memoria locale**: è una memoria dedicata per ogni thread a cui può accedere solo esso, non mantiene lo stato tra due kernel

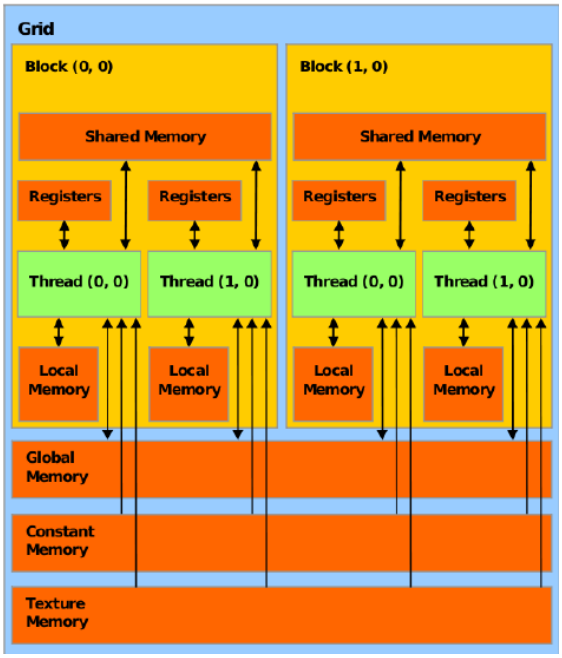

Figura 1.5: Struttura e gerarchia della memoria in una GPU CUDA.
Fonte: `https://www.researchgate.net/figure/CUDA-memory-model-from-7_fig1_4373162`

1.4.3 Buone tecniche di gestione della memoria

Come è mostrato nella sezione precedente, CUDA prevede varie tipologie di memoria per cercare di ottimizzare il rapporto tra unità di esecuzione e memoria; infatti, come abbiamo già potuto notare, questo rappresenta un forte limite per le applicazioni. In particolare, la memoria che più delle altre può aiutare molto l'esecuzione è la memoria condivisa. Questa memoria, avendo un accesso molto più rapido, se sfruttata può garantire un forte impatto nel migliorare le prestazioni; infatti, possiamo considerarla come una sorta di cache gestita manualmente dal programmatore, che permette anche di diminuire gli accessi alla memoria globale. Un particolare meccanismo che offre CUDA è il raggruppamento delle letture, ovvero, nel caso in cui i thread di un warp necessitano di leggere locazioni di memoria adiacenti, l'hardware è ottimizzato per eseguirle in una singola operazione. Questa possibilità permette di ottenere notevoli miglioramenti delle prestazioni; infatti, si riducono gli accessi alla memoria e si sfrutta al massimo il bus dati, riducendo la quantità di dati inutili trasferiti. Per ottenere ciò è necessario progettare strutture dati adeguate al modo in cui verranno utilizzate.

Capitolo 2

Il Data Distribution Management

In questo capitolo, vedremo il contesto in cui si inserisce e cos'è il **Data Disitribution Management** (DDM), qual è il problema principale che affronta e infine analizzeremo lo stato dell'arte nella risoluzione di tale problema.

2.1 High Level Architecture

Uno strumento molto diffuso al giorno d'oggi sono le simulazioni, le quali permettono di riprodurre scenari reali in un ambiente virtuale. Alcuni possibili esempi di ambiti in cui si utilizzano abitualmente sono: studio di situazioni di emergenza [11], i trasporti e la logistica [8], o sistemi biologici complessi [13].

In particolare, si sta cercando di modellare fenomeni sempre più complessi con un numero crescente di entità, sposso di diversa natura e in sistemi distribuiti. Perciò abbiamo la necessità di avere punti di riferimento che diano ordine. In questo senso, sono state definite delle linee guida per la modellazione delle simulazioni [15], tuttavia rimane il problema di come affrontare la crescita della complessità.

Per cercare di rispondere, almeno in parte, a questo problema, il Dipartimento della Difesa degli Stati Uniti ha contribuito a realizzare uno standard chiamato **High Level Architecture** (HLA). Questo standard definisce una architettura per sistemi di simulazione distribuiti, gestendo i problemi di interoperabilità e permettendo a diverse simulazioni di interagire indipendentemente dalla piattaforma sulla quale risiedono [2].

HLA prevede che le entità presenti in una simulazione, chiamate federati, interagiscano attraverso una interfaccia standard, che permette di nascondere le peculiarità di ogni sistema, ma far sì che ogni federato veda tutti gli altri allo stesso modo. In questo modo si possono realizzare grandi modelli composti da entità eterogenee. L'implementazione dell'interfaccia standard è affidata alla Run Time Infrastructure (RTI) [1], la quale definisce vari servizi, tra i quali il DDM che è

responsabile della comunicazione tra federati. Per lo scambio di informazioni HLA prevede che si segua la struttura e la logica definita dall'Object Model Template (OMT) [3].

2.2 DDM

Il Data Distribution Management comprende vari servizi atti a ridurre la quantità di dati non necessari da trasferire tra i federati, diminuendo così sia il traffico sulla rete, sia il carico di lavoro dei federati riceventi (che devono scartare i dati non necessari). Infatti, è importante che vi sia un filtro nello scambio delle informazioni, perché sarebbe inutile che un federato venga a conoscenza di informazioni circa eventi a cui non è interessato, facendo circolare informazioni inutili sulla rete che potrebbero causarne un sovraccarico. Ad esempio, in un videogioco multigiocatore che simula uno scenario di guerra, ogni giocatore sarà interessato a ricevere informazioni circa gli spari intorno a sé, mentre non è interessato a ciò che succede tra altri giocatori in posizioni lontane sulla mappa.

Il DDM per far in modo che siano trasmesse solo informazioni realmente utili, permette ad ogni federato di specificare i propri interessi per cui desidera ricevere notifiche. Per realizzare ciò HLA prevede che il mondo virtuale da simulare sia mappato in un sistema di coordinate multidimensionale. Gli interessi dei federati sono rappresentati da regioni rettangolari in d dimensioni in questo spazio. In particolare, si denotano due tipi di regione:

- *subscription region*: astrazioni dell'insieme dei dati del mondo per cui un'entità è interessata a ricevere aggiornamenti
- *update region*: astrazioni dell'insieme dei dati del mondo per cui un'entità è interessata a trasmettere gli aggiornamenti

Il DDM ha come compito principale quello di riconoscere e riportare l'insieme delle coppie subscription-update region che si intersecano, questa operazione è nota in letteratura come matching.

2.3 Definizione del problema

Vediamo una definizione formale del problema del matching che deve risolvere il DDM.

Problema del matching. Dati due insiemi $S = S_1, \ldots, S_n$ e $U = U_1, \ldots, U_m$ di rettangoli in d dimensioni: enumerare tutte le coppie $(S_i, U_j) \subseteq S \times U$ tali che

$S_i \cap U_j \neq \emptyset$; ogni coppia deve essere riportata una sola volta senza un particolare ordinamento.

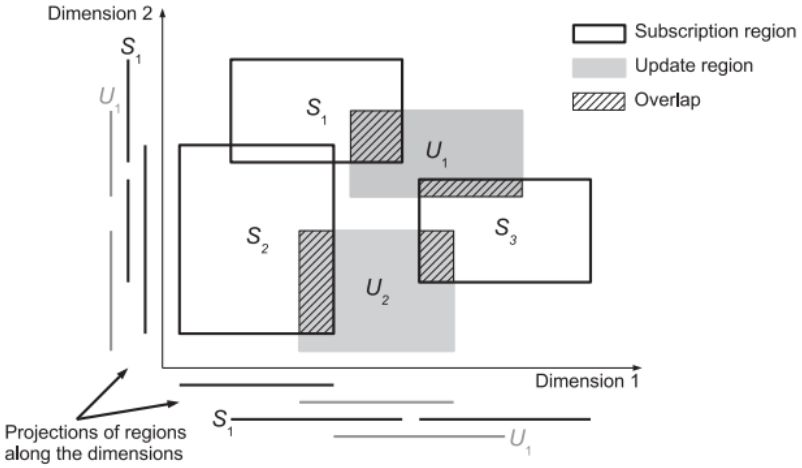

Figura 2.1: Problema del matching in d=2 dimensioni.
Fonte: `https://doi.org/10.1145/3369759`

Nella figura 2.1 si può vedere un esempio di questo problema in $d = 2$ dimensioni con tre subscription region $\mathbf{S} = \{S_1, S_2, S_3\}$ e due update region $\mathbf{U} = \{U_1, U_2\}$. In questo esempio ci sono quattro coppie di intersezioni subscription-update e sono: $\{(S_1, U_1), (S_2, U_2), (S_3, U_1), (S_3, U_2)\}$.

Possiamo fare alcune considerazioni per risolvere questo problema. In particolare, per quel che riguarda le coordinate delle varie regioni, sarebbe sufficiente considerarle come numeri interi come è previsto in HLA; tuttavia, solitamente si cerca di risolvere il problema in un contesto più generale prevedendo la possibilità che siano numeri reali. Un'altra considerazione che viene presa nella risoluzione di questo problema è sulle dimensioni. Infatti, nonostante il problema sia definito in d dimensioni, si tende a cercare soluzioni per il caso monodimensionale. Perché nel caso $d > 1$ è possibile considerare le proiezioni delle varie regioni su ogni asse, risolvere d problemi in una dimensione, e poi trovare la soluzione per tutte le dimensioni intersecando i risultati dei vari problemi più semplici.

2.4 Principali algoritmi nel DDM

L'operazione del matching può risultare decisamente costosa, infatti negli anni sono stati studiati numerosi algoritmi per renderla più efficiente. In particolare, ultimamente visto l'aumentare delle dimensioni delle simulazioni, si stanno cercando anche soluzioni parallele che riducano i tempi di esecuzione. Vedremo ora

alcuni degli algoritmi più conosciuti per risolvere questo problema, analizzando anche versioni parallele.

2.4.1 Brute Force

Un primo possibile algoritmo per risolvere questo problema può essere quello che controlla tutte le $n \times m$ possibili coppie subscription-update (s, u), verificando per ognuna se le rispettive regioni si intersecano. Questa soluzione viene chiamata **Brute Force Matching** (BFM) e ha un costo computazionale che è proporzionale al numero delle coppie, cioè $\Theta(nm)$. La semplicità dell'algoritmo viene pagata nei tempi di esecuzione. Nel caso in cui volessimo realizzare una versione parallela di questo algoritmo è possibile farlo semplicemente affidando una parte di tutte le possibili coppie a ciascun processore a disposizione. Sia P il numero di processori, avremo un costo per ciascuno pari a $\Theta(nm/P)$.

2.4.2 Grid Base Matching

Un possibile miglioramento di BFM è l'algoritmo proposto da Boukerche e Dzermajko [6], il **Grid Base Matching** (GBM). L'algoritmo GBM prevede di suddividere lo spazio multidimensionale, che rappresenta la simulazione, in tante celle a d dimensioni, formando una griglia. Si associa ogni subscription e update region alle celle con cui vi è una sovrapposizione e quando una update region genera un evento, questo verrà inviato a tutte le subscription region che condividono una stessa cella della griglia. Tuttavia, questo approccio genera una soluzione non ottimale, infatti non è detto che se due regioni condividono una cella siano anche sovrapposte. Per eliminare tra le coppie di regioni rilevate quelle che non si intersecano realmente, si applica ad ogni cella della griglia il metodo BFM [22].

Supponendo uniformità nella distribuzione delle regioni all'interno della griglia, avremo che ogni cella conterrà n/n_celle subscription region e m/n_celle update region. Quindi l'approccio BFM applicato in ogni cella richiederà $O(nm/n\_celle^2)$, avendo un costo totale complessivo nel caso pessimo pari a $O(nm/n\_celle)$. In conclusione, questo approccio permette di ridurre i costi rispetto a BFM di un fattore n_celle. Tuttavia, nella pratica è difficile trovare un numero di celle adeguato, infatti idealmente potremmo pensare di aumentare il numero di celle che essendo al denominatore ridurrebbe i costi, però, in questo modo ogni regione sarà associata ad un numero sempre più grande di celle, il che provoca un aumento delle computazioni necessarie alla gestione della griglia.

2.4.3 Sort Base Matching

Una soluzione che si è dimostrata efficiente per il DDM è l'algoritmo **Sort Base Matching** (SBM) [14, 20]. Questo algoritmo lavora con segmenti, ovvero regioni con $d = 1$ dimensioni. L'implementazione prevede di preparare una lista contenente le estremità di tutte le regioni (subscription e update), ordinarla in ordine crescente, poi scorrerla tenendo traccia in ogni punto delle regioni per cui è stato visitato l'estremo inferiore ma non quello superiore, distinguendole per tipologia. Nel momento in cui scorrendo la lista si incontra un estremo superiore, è possibile ottenere le intersezioni ritornando le coppie composte dalla regione a cui appartiene l'estremo e quelle presenti nell'insieme delle regioni parzialmente visitate dell'altra tipologia.

Il costo computazionale di questo algoritmo è $O(N \log N + K)$. Fissato $N = n + m$ il numero totale di segmenti, abbiamo che il costo per l'ordinamento ($O(N \log N)$) domina il costo della scansione della lista ordinata, mentre rimane $O(K)$ che rappresenta il costo per riportare le K intersezioni.

Questo algoritmo si è dimostrato, nella pratica, essere molto efficiente, con lo svantaggio che non appare facilmente parallelizzabile. Tuttavia Marzolla e D'Angelo sono riusciti a fornire una versione parallela per architetture multiprocessore a memoria condivisa [17]. Inoltre, nei prossimi capitoli analizzeremo un possibile approccio per GPU CUDA.

Capitolo 3

Algoritmo CUDA

In questo capitolo presentiamo un possibile algoritmo parallelo, implementabile in CUDA, per la ricerca delle intersezioni tra segmenti presenti in un insieme, basato sull'algoritmo Sort Base Matching descritto nel capitolo 2.
Questo problema rappresenta una semplificazione rispetto al problema in cui si cercano le intersezioni tra segmenti di due insiemi distinti, ma costituisce comunque la parte fondamentale di tale problema: è infatti possibile costruire l'algoritmo completo estendendo l'algoritmo che risolve questo problema semplificato.

3.1 Come realizzare un algoritmo parallelo

Quando abbiamo un algoritmo seriale da parallelizzare, innanzitutto dobbiamo capire se è possibile farlo; infatti, nel caso in cui presenti una struttura intrinsecamente seriale, la versione parallela sarà da ripensare totalmente partendo da zero.

Nel caso in cui abbiamo un algoritmo seriale che può essere parallelizzato, si può pensare di costruire la versione parallela incrementalmente. Il punto di partenza sarà l'analisi dell'algoritmo seriale orientata a capire come suddividere i calcoli necessari alla risoluzione del problema, tra più unità di elaborazione. In particolare, è utile riconoscere la presenza di forme algoritmiche ricorrenti, dette pattern architetturali, per le quali sono note alcune tecniche risolutive.

Nel primo passaggio si evidenziano tra le varie porzioni dell'algoritmo seriale quelle pronte per essere parallelizzate e quelle che presentano criticità. Per fare ciò si cercano eventuali dipendenze tra le istruzioni; nel caso in cui non siano rilevate siamo in presenza del pattern *embarrassingly parallel*, il quale ci suggerisce che è possibile suddividere il calcolo tra più unità di elaborazione indipendenti. Nel caso in cui siano state riscontrate dipendenze, è necessario analizzarle nei dettagli, cercando di capire se è possibile applicare opportune strategie per eluderle, oppure se

le istruzioni presentano un comportamento intrinsecamente seriale. Un elemento importante da tenere in considerazione negli algoritmi paralleli è la sincronizzazione tra le varie unità di elaborazione. Infatti, bisogna che la sincronizzazione sia ben progettata, perché non essendoci determinismo nei flussi di esecuzione paralleli, se questo passaggio non venisse eseguito correttamente potrebbero presentarsi comportamenti anomali difficili da individuare.

Possiamo dire quindi che per realizzare un algoritmo parallelo è necessario avere la conoscenza delle tecniche da utilizzare e l'esperienza per utilizzare opportuni accorgimenti.

3.2 Analisi dell'algoritmo seriale

Analizziamo l'algoritmo seriale che individua le intersezioni all'interno di un insieme di segmenti (Algoritmo 1). L'algoritmo prevede di prendere in input un insieme di segmenti, individuare le intersezioni presenti tra di essi e restituirle come output. Ogni segmento dell'input è definito come un intervallo di cui si conoscono gli estremi.

Algoritmo 1 SBM seriale in un insieme

```
1: T ← ∅
2: for all segmenti x ∈ S do
3:     Inserisci x.lower e x.upper in T
4: Ordina T in ordine crescente
5: SubSet ← ∅
6: for all estremo t ∈ T in ordine crescente do
7:     if t è l'estremo inferiore di s then
8:         SubSet ← SubSet ∪ {s}
9:     else
10:        SubSet ← SubSet \ {s}
11:        for all u ∈ Subset do REPORT(s, u)
```

L'idea che sta alla base dell'algoritmo è quella di ordinare gli estremi dei segmenti come se li ponessimo tutti lungo un asse cartesiano, per poi scorrere in ordine crescente gli estremi, individuando le intersezioni.
In particolare, per individuare le intersezioni, si tiene traccia, in ogni punto raggiunto durante lo scorrimento ideale dell'asse cartesiano, dei segmenti di cui è stato visitato l'estremo inferiore, ma non quello superiore. Questo ci permette di dire, nel momento in cui si visita l'estremo superiore di un segmento, con chi si interseca il segmento appena chiuso. Nella figura 3.1 è possibile vedere un esempio

di esecuzione dell'algoritmo; guardando la figura da sinistra verso destra si simula il comportamento delle iterazioni che portano ad individuare le intersezioni. In particolare, si può notare come evolve l'insieme che tiene traccia dei segmenti parzialmente visitati.

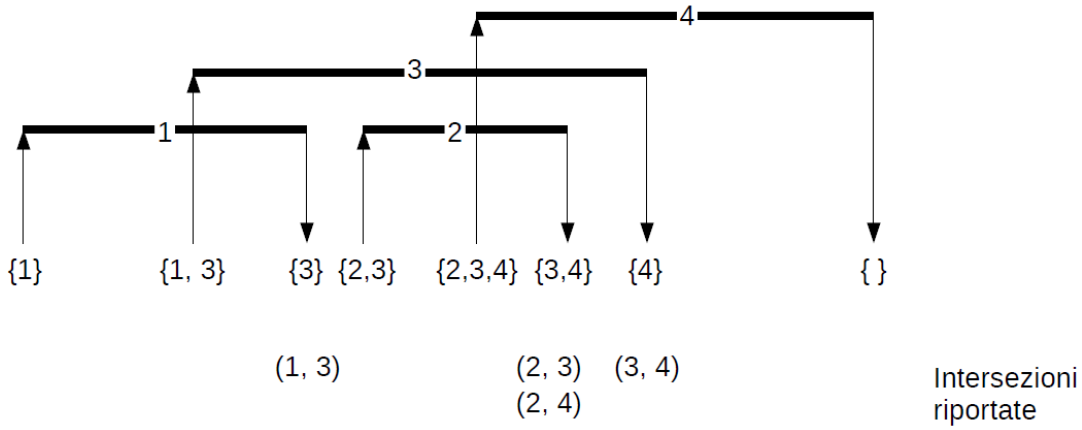

Figura 3.1: Esempio di esecuzione dell'algoritmo seriale

Possiamo quindi vedere l'algoritmo diviso principalmente in due fasi: la prima in cui viene costruita e ordinata una lista contenente tutti gli estremi dei segmenti, la seconda in cui si individuano tutte le intersezioni.

3.3 Definizione dell'algoritmo parallelo

Vediamo come è possibile realizzare una versione parallela adatta ad essere implementata in CUDA. Per fare ciò analizziamo le due fasi che compongono l'algoritmo alla ricerca di dipendenze che potrebbero ostacolare la realizzazione di una versione parallela.

Per quel che riguarda la prima fase non si evidenziano particolari dipendenze, in particolare la costruzione della lista degli estremi, se implementata con un array di $2 \times N$ elementi, invece che una lista concatenata, è una computazione embarrassingly parallel, in cui possiamo assegnare ogni singola posizione ad una unità di esecuzione differente. L'ordinamento degli estremi può essere anch'esso parallelizzato: è presente infatti una ampia letteratura su questo tema, esistono algoritmi paralleli ottimi per architetture a memoria condivisa, come mergesort [9] e quicksort [23], ma anche numerose implementazioni per GPU quali ad esempio: radix sort [18], mergesort [10], quicksort [7], samplesort [16] e approcci ibridi [21].

Analizzando la seconda fase è possibile notare la presenza di alcune criticità che potrebbero impedire una esecuzione in parallelo. In particolare, l'algoritmo prevede di iterare in modo ordinato la lista degli estremi tenendo traccia opportunamente dei segmenti visitati, ma questo si scontra con la nostra volontà di eseguire le iterazioni in parallelo. Infatti, se lavorassimo su più estremi in parallelo, non avremmo più il controllo sull'ordine in cui vengono realizzate le iterazioni, causando un comportamento sbagliato dell'algoritmo. Siamo infatti in presenza di un problema detto *loop-carried dependencies*, ovvero dipendenze sui dati, per cui il risultato dell'iterazione corrente dipende da ciò che è successo nelle iterazioni precedenti. In questo caso la dipendenza è sull'insieme dei segmenti aperti, il cui valore dipende dalla computazione delle iterazioni precedenti.
Per risolvere questo problema, è necessario trovare un metodo parallelizzabile per ottenere lo stesso risultato. In particolare, dobbiamo riuscire a ricreare il comportamento dell'insieme dei segmenti parzialmente visitati alle varie iterazioni. Infatti, conoscendo lo stato dell'insieme alle varie iterazioni, si possono ricavare le intersezioni guardando i segmenti presenti nell'insieme nel momento in cui si visita un estremo superiore. Vediamo di seguito una possibile strategia.

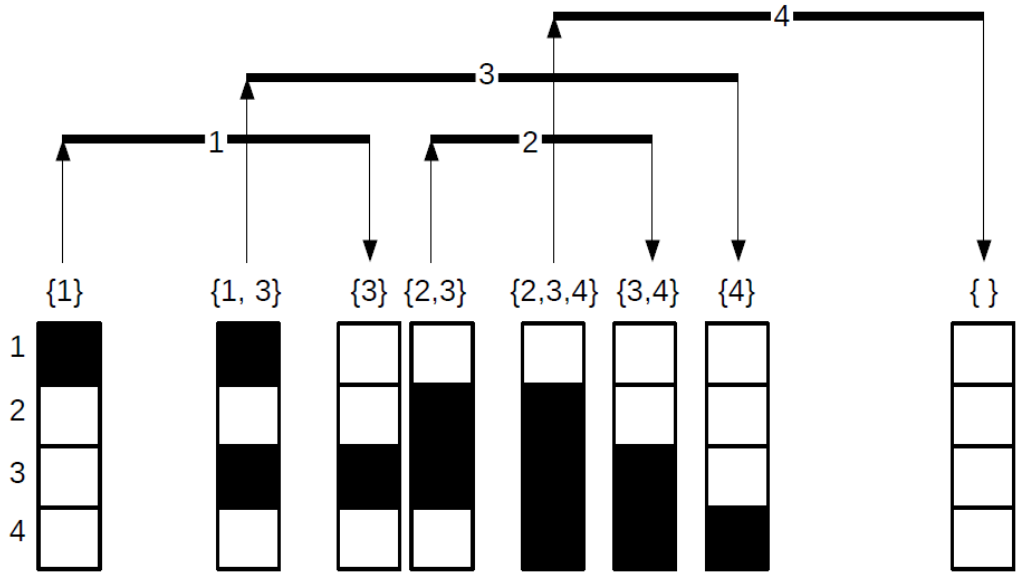

Figura 3.2: Esempio della rappresentazione dell'insieme dei segmenti attraverso bitmap

L'idea è quella di avere per ogni estremo l'istantanea dell'insieme al termine dell'iterazione che lo ha elaborato: per fare ciò possiamo utilizzare delle bitmap,

ovvero array di N elementi in cui ogni segmento è associato ad una posizione con due possibili valori 0 e 1, indicanti l'assenza o la presenza nell'insieme, nella figura 3.2 si può vedere un esempio.

Per ottenere questo risultato attraverso una computazione parallela consideriamo per semplicità tutte le bitmap unite come se fossero le colonne di una matrice di bit. In questa matrice l'indice di riga rappresenta l'identificativo del segmento e l'indice di colonna l'iterazione che ha generato la bitmap. Si vuole ottenere una matrice in cui ogni elemento in posizione (i, j) ci dice se il segmento i è tra i segmenti parzialmente visitati dopo aver eseguito l'iterazione j, ovvero dopo l'elaborazione dell'estremo nella j-esima posizione della lista.
Questa matrice si può ottenere partendo da due matrici di bit. La prima in cui la posizione (i, j) indica se il segmento i all'iterazione j potrebbe essere presente nell'insieme, poiché sappiamo che è stato visitato l'estremo sinistro all'iterazione k dove $k \leq j$, ma non possiamo fare ipotesi sull'estremo destro. La seconda in cui la posizione (i, j) indica se il segmento i all'iterazione j è sicuramente assente nell'insieme, poiché sappiamo che è stato visitato l'estremo destro all'iterazione w dove $w \leq j$. La costruzione di queste matrici si fonda sull'ipotesi che gli estremi nella lista siano ordinati in ordine crescente; perciò, per ogni segmento l'estremo inferiore viene visitato prima del superiore, quindi $k < w$. Inoltre, possiamo affermare per la stessa ipotesi e per costruzione che: l'insieme delle posizioni a 1 della seconda matrice sono un sottoinsieme di quelle della prima.

Dalla costruzione delle due matrici è possibile ottenere la matrice desiderata applicando l'operatore XOR tra le celle delle due matrici. In particolare, sono certo che l'operatore XOR restituirà 1 solo nelle posizioni in cui l'intervallo è stato aperto (1 nella prima matrice), ma non ancora chiuso (0 nella seconda matrice). Infatti, per l'assunzione fatta in precedenza, non capiterà mai il caso in cui non so se l'intervallo è stato aperto (0 nella prima matrice), ma so che è stato chiuso (1 nella seconda matrice). Nella figura 3.3 è possibile vedere graficamente un esempio, si può notare anche che il risultato ottenuto è lo stesso di quello desiderato (Fig. 3.2).

Il processo descritto è completamente parallelizzabile in due fasi: la prima in cui si preparano le matrici e la seconda in cui si uniscono i risultati.
Nella prima fase è possibile lavorare contemporaneamente su ogni riga delle due matrici. In particolare, è necessario per ogni riga individuare la cella in cui compare il primo 1, per poi propagarlo fino all'ultima colonna. Per individuare il primo 1 su ogni riga, possiamo assegnare un estremo ad ogni unità di esecuzione. Quest'ultima conoscendo: il tipo di estremo, l'identificativo del segmento e la posizione dell'estremo nella lista; può realizzare la computazione. Infatti, con queste informazioni è possibile assegnare il valore 1 alla cella definita da: il tipo di estremo, inferiore o superiore, che indica quale matrice utilizzare, l'identificativo

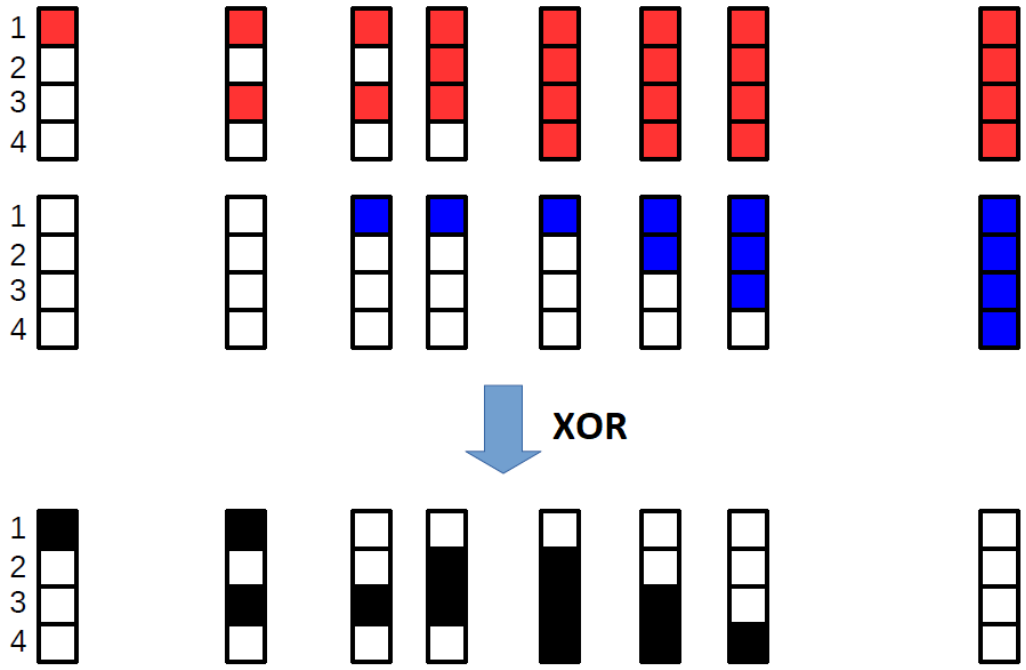

Figura 3.3: Esempio applicazione xor tra bitmap

del segmento e la posizione nella lista che indicano rispettivamente la riga e la colonna in cui assegnare il valore.

Per propagare il valore 1 dalla posizione settata fino all'ultima colonna della matrice, possiamo pensare di applicare l'operatore OR tra un elemento e il suo predecessore: in questo modo appena si individua un 1 esso si propagherà fino all'ultimo elemento. Questo tipo di operazione si può eseguire in parallelo in modo efficiente: si tratta del pattern architetturale *scan*, noto anche come *computazione parallela prefissa* o *somme prefisse*.

La scan prevede che ci sia una sequenza di elementi $x_0, ..., x_{N-1}$ con $N > 0$ e un operatore binario associativo $\oplus$ da applicare tra le coppie consecutive. Sono presenti due tipi di computazione prefissa, *scan inclusiva* e *scan esclusiva*, i quali producono due nuove sequenze $y_0, \ldots, y_{N-1}$ (scan inclusiva) e $z_0, \ldots, z_{N-1}$ (scan esclusiva) definite nel seguente modo:

$$y_j = \begin{cases} x_0 & \text{se } j = 0 \\ y_{j-1} \oplus x_j & \text{se } j > 0 \end{cases} \qquad z_j = \begin{cases} 0 & \text{se } j = 0 \\ z_{j-1} \oplus x_j & \text{se } j > 0 \end{cases}$$

in cui 0 rappresenta l'elemento neutro dell'operatore $\oplus$, per cui vale la proprietà

$0 \oplus x = x$.
Per eseguire questo tipo di operazione esistono vari algoritmi paralleli, ad esempio Hillis e Steele [12] hanno proposto un algoritmo per eseguire la scan di N elementi con N processori in $O(\log N)$ per ogni esecuzione parallela, una quantità di calcolo totale pari a $O(N \log N)$. Questo risultato è stato migliorato da Blelloch [5], il quale ha descritto una versione parallela su N elementi utilizzando P processori che richiede un tempo pari a $O(N/P + \log P)$. Quest'ultimo algoritmo risulta ottimo quando $N > P \log P$, infatti in tal caso la quantità totale di calcolo necessario è pari a quella necessaria per l'algoritmo seriale, cioè $O(N)$.

Quindi, possiamo costruire la matrice che descrive l'evoluzione dell'insieme dei segmenti parzialmente visitati, partendo dalla costruzione, in parallelo, di due matrici come descritto in precedenza. Nelle figure 3.4 e 3.5 è possibile vedere graficamente attraverso un esempio come ciò avviene.
Il passaggio successivo alla costruzione delle due matrici, ovvero l'esecuzione dello XOR tra le celle di tali matrici, è una computazione embarrassingly parallel, in cui si può eseguire l'operazione contemporaneamente su tutte le celle della matrice.

In conclusione, si è ottenuto un algoritmo parallelo, implementabile su GPU, per calcolare le intersezioni tra segmenti in un insieme, che possiamo riassumere in pseudocodice nell'algoritmo 2.

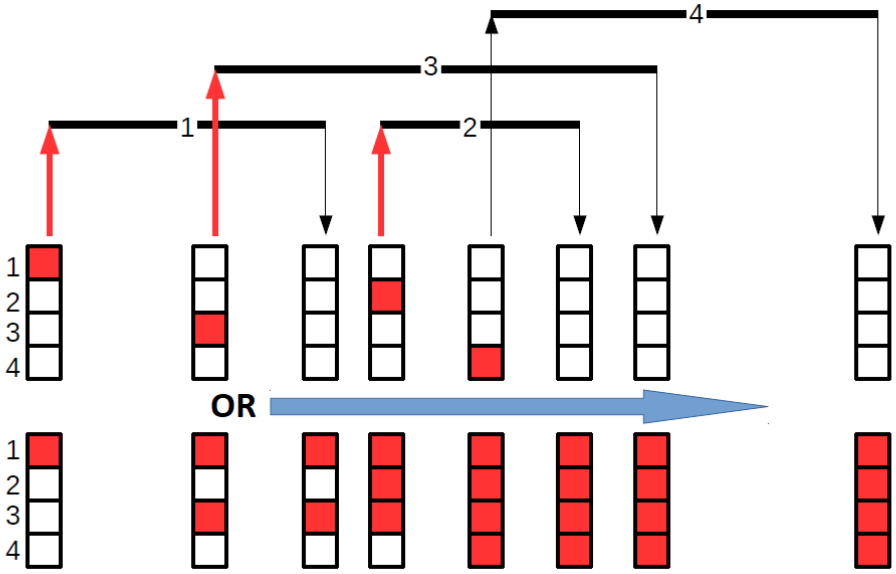

Figura 3.4: Costruzione delle bitmap riferite agli estremi inferiori

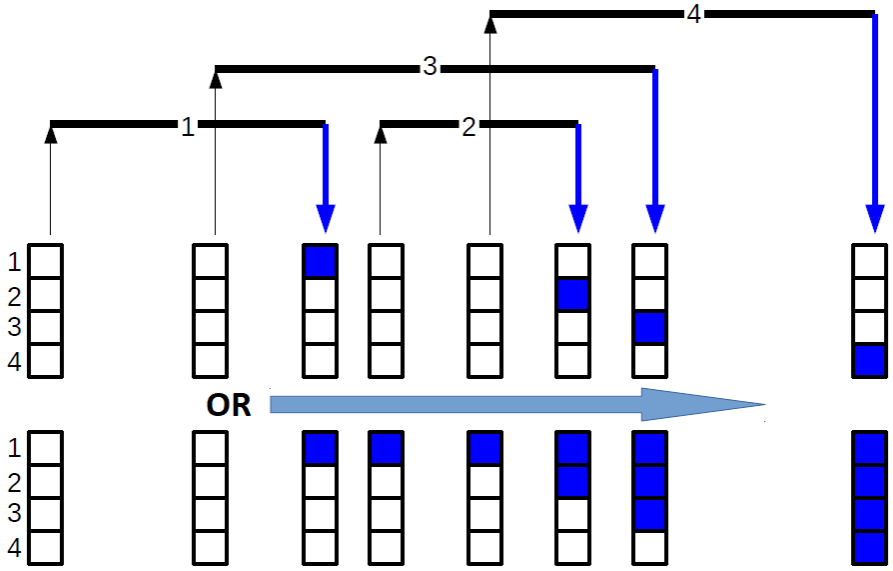

Figura 3.5: Costruzione delle bitmap riferite agli estremi superiori

Algoritmo 2 SBM parallelo in un insieme

```
1: T ← array vuoto [0...N − 1] di estremi
2: for all segmenti x ∈ S in parallelo do
3:     Inserisci x.lower e x.upper in T
4: Ordina T in ordine crescente in parallelo
5: LowMatrix ← matrice [0...N − 1, 0...2N − 1] di bit a 0
6: UpMatrix ← matrice [0...N − 1, 0...2N − 1] di bit a 0
7: for j ← 0, 2N − 1 in parallelo do
8:     t ← T[j]
9:     i ← t.id
10:    if t è l'estremo inferiore di s then
11:        Mat ← LowMatrix
12:    else
13:        Mat ← UpMatrix
14:    Mat[i, j] = 1
15:    Scan inclusiva usando l'operatore OR di Mat sulla riga i in parallelo
16: for i ← 0 to N − 1 in parallelo do
17:    for j ← 0 to N − 1 in parallelo do
18:        IntersectionMatrix[i, j] ← LowMatrix[i, j] XOR UpMatrix[i, j]
19: for all estremi superiori t in T in parallelo do
20:    for i ← 0, N do
21:        if IntersectionMatrix[i, j] = 1 then
22:            REPORT(j,i)
```

Capitolo 4

Implementazione

In questo capitolo analizzeremo una possibile implementazione dell'algoritmo descritto nel capitolo precedente. In particolare, vedremo come è possibile implementarlo utilizzando il linguaggio CUDA C/C++ e la libreria Thrust.

4.1 La libreria Thrust

Thrust è una libreria sviluppata da NVIDIA per facilitare la programmazione CUDA [4]. In particolare, fornisce attraverso un'interfaccia astratta, varie funzionalità tra cui: i principali algoritmi per la programmazione parallela, funzioni per la gestione della memoria e alcune strutture dati. Componendo in modo opportuno questi elementi è possibile implementare algoritmi complessi scrivendo codice compatto e leggibile.

Il principale vantaggio che fornisce questa libreria è quello di aumentare la produttività del programmatore; infatti, permette di definire ad alto livello cosa implementare senza pensare al come, sarà la libreria automaticamente ad attuare la soluzione più efficiente. Questo è molto importante soprattutto nella programmazione CUDA in cui si lavora spesso a basso livello: con la necessità di gestire le varie memorie e organizzare le esecuzioni in termini di griglia e blocchi.

Un altro vantaggio che deriva dall'astrazione fornita è la robustezza; infatti, si riescono a superare limiti legati ad hardware specifici, e viene garantita una corretta esecuzione in tutte le GPU CUDA. Thrust, inoltre, si è rivelata in alcuni casi d'uso reali utile al miglioramento delle prestazioni. Infatti, alcune funzionalità di uso comune vengono appositamente ottimizzate, ad esempio l'operazione di ordinamento dei dati di tipo primitivo. Tuttavia, in altri casi astrarre potrebbe essere uno svantaggio.

La libreria è scritta in CUDA C/C++ e garantisce l'interoperabilità con tutto l'ecosistema CUDA. In particolare, è possibile scegliere il giusto livello di astrazione

da utilizzare per specifiche implementazioni, combinando Thrust e CUDA C. In questo modo si cerca di trovare il giusto equilibrio tra l'ottimizzazione di basso livello e la semplicità di programmazione.

4.2 Implementazione

4.2.1 Strutture dati

Per implementare l'algoritmo che calcola le intersezioni all'interno di un insieme di segmenti, abbiamo definito alcune strutture dati. In particolare, per rappresentare i dati del dominio abbiamo definito le strutture *interval* e *endpoint*, che rappresentano rispettivamente un segmento e una sua estremità. Interval contiene semplicemente le coordinate monodimensionali degli estremi. Endpoint contiene tutte le informazioni di una estremità quali: l'id del segmento, il valore della coordinata che rappresenta, il tipo di estremità (inferiore o superiore), e infine è stato predisposto il campo che rappresenta la tipologia di segmento (subscription o update). Per questa struttura è stato ridefinito l'operatore $<$ necessario per l'implementazione dell'ordinamento.

La struttura dati principale che è stata definita per l'implementazione CUDA dell'algoritmo è *device_bitmatrix*, la quale rappresenta la matrice su cui lavorare in parallelo per individuare le intersezioni. Questa struttura è stata progettata affinché potesse risiedere nella memoria del device, in modo tale che si riduca lo scambio di dati tra le memorie di host e device. Questo accorgimento è una buona pratica da considerare quando si realizzano programmi CUDA. Infatti, lo scambio di dati risulta essere una computazione che fa perdere tempo nell'esecuzione dell'algoritmo, inoltre vista la differenza di velocità tra processori e memorie, può generare colli di bottiglia che penalizzano le prestazioni.

La matrice idealmente dovrebbe memorizzare dei bit, in realtà è stata implementata per memorizzare degli interi senza segno a 8 bit, questo è stato fatto perché è il tipo di dato predefinito con la minore occupazione di memoria e permette l'accesso concorrente alle varie celle; in realtà come vedremo in seguito questa struttura comporta dei forti limiti.

4.2.2 Algoritmo

Analizziamo ora come è stato implementato l'algoritmo commentando eventuali scelte implementative, in particolare come è stato sfruttato il parallelismo. L'implementazione sfrutta sia la libreria Thrust, sia specifiche funzioni kernel. Nella figura 4.1 si può vedere un diagramma di sequenza che descrive l'implementazione

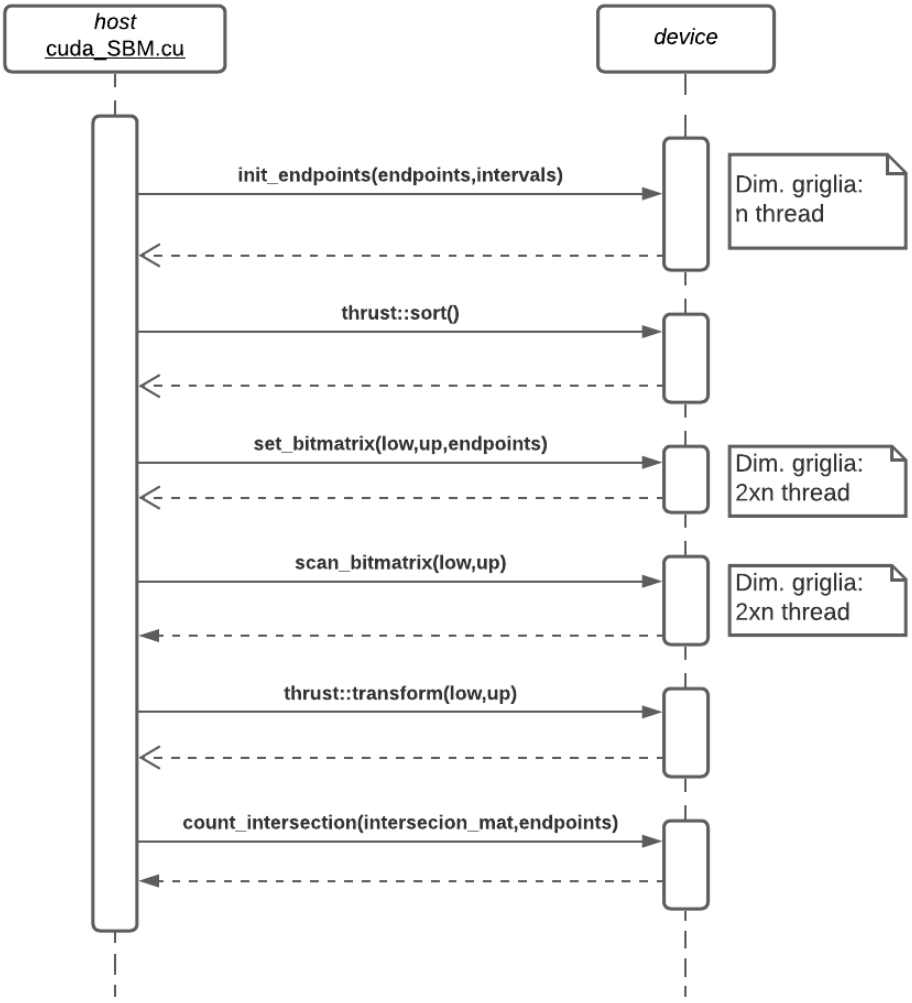

Figura 4.1: Diagramma di sequenza che descrive l'implementazione dell'algoritmo

dell'algoritmo in cui si può osservare l'nterazione tra host e device attraverso le principali funzioni.

Il primo passaggio che prevede l'algoritmo è inserire gli estremi in un vettore e poi ordinarli. Per inserire gli estremi in un vettore è stata scritta una funzione kernel che affida ogni segmento ad un thread il quale inserisce nel vettore sia l'estremo destro che quello sinistro. C'era la possibilità di affidare ogni estremo ad un thread differente, però in tal caso sarebbe stato necessario inserire un costrutto `if-else` per capire quale estremo considerare. Tuttavia, visto che all'interno di un warp vengono eseguite istruzioni SIMD, nel caso in cui si dovessero eseguire entrambi i rami ogni thread è come se trattasse due estremi: quindi lanciare un numero doppio i thread potrebbe risultare controproducente. L'ordinamento del

vettore è stato realizzato sfruttando la funzione parallela fornita dalla libreria.

Per individuare le intersezioni, come prevede l'algoritmo, sono state utilizzate due strutture device_bitmatrix. Queste vengono gestite attraverso alcuni kernel eseguiti da un numero di thread pari al numero di estremi. Per eseguire l'operazione di scan si sfrutta la funzione `inclusive_scan` di Thrust. L'operazione di XOR tra le celle delle due matrici, per formare quella in cui ricavare le intersezioni, viene svolta utilizzando la funzione `transform` di Thrust, che applica in parallelo un operatore binario tra due input.

In questa implementazione non vengono riportate le intersezioni, anche se sarebbe possibile farlo, ma ci si limita a conoscerne il numero. Per realizzare ciò bisogna lavorare sugli estremi superiori, ma non conoscendo in quali posizioni si trovano, si assegna un estremo ad ogni thread e solo quelli a cui è stato affidato un estremo superiore eseguiranno la computazione. Ogni thread conta nella colonna corrispondente al suo estremo il numero di celle a 1, il risultato viene memorizzato in un vettore che conterrà i risultati parziali, infine la somma di quest'ultimo vettore viene eseguita sfruttando la funzione Thrust `reduce`, che esegue una riduzione. L'operazione di riduzione rappresenta un pattern di programmazione parallela che viene applicato quando da un elenco di elementi si vuole ottenere un unico elemento che rappresenta il risultato di una operazione applicata tra tutti. Un esempio di applicazione tipico è proprio la somma di un vettore.

4.3 Limiti di memoria

La soluzione implementata mostra un forte limite nell'utilizzo della memoria. Questo deriva dal fatto che le matrici che si utilizzano per individuare le intersezioni occupano uno spazio quadratico rispetto al numero di segmenti; infatti, le matrici hanno un numero di righe pari al numero di segmenti e un numero di colonne pari al numero di estremità cioè il doppio del numero di segmenti. L'implementazione prevede che si utilizzi un byte per ogni cella e si utilizzino due matrici dando luogo ad una occupazione di memoria pari a $4 \times n^2$ byte. Perciò, considerando una memoria di 8GB, questa sarebbe esaurita con meno di 50000 segmenti. Questo ci fa capire che la soluzione non è facilmente scalabile su larga scala. Inoltre, questo utilizzo massiccio di memoria influisce negativamente sulle prestazioni che risultano pessime e non competitive con altre.

4.4 Analisi delle prestazioni

Per analizzare le prestazioni di programmi per GPU non è possibile basarsi su metriche comunemente utilizzate per la valutazione di programmi per CPU. In

particolare, non è possibile utilizzare metriche che operano in funzione del numero di processori. Infatti, nelle GPU non abbiamo un controllo diretto sull'hardware e inoltre il numero di thread che mettiamo in esecuzione dipende fortemente dalla dimensione del problema. Una metrica che si è soliti utilizzare per valutare le prestazioni su GPU è il throghput, ovvero il numero di elementi processati per unità di tempo. In particolare, si osserva la variazione del throughput all'aumentare della dimensione dell'input. Solitamente si può osservare come il throughput aumenta con l'aumentare dell'input fino a raggiungere un certo limite. Questo comportamento è dovuto al fatto che all'aumentare dei dati si sfrutta sempre meglio il parallelismo offerto dalla GPU.

Nel nostro caso (Fig. 4.2), possiamo vedere come ciò venga confermato. Bisogna però notare uno strano comportamento nella fase iniziale, ovvero abbiamo un brusco calo del throughput a 7000 segmenti, questo probabilmente è dovuto al fatto che viene meno una ottimizzazione. Questa perdita di throughput viene recuperata solamente aumentando le dimensioni dell'input arrivando intorno a 19000 segmenti.

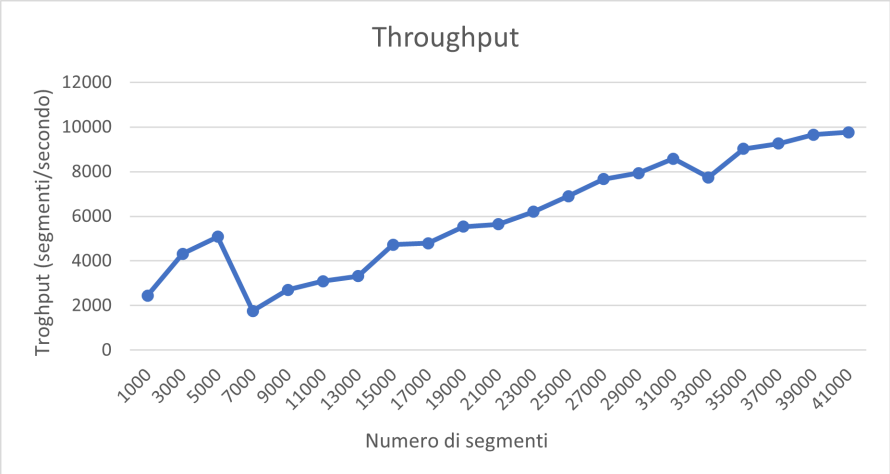

Figura 4.2: Throughput in funzione della dimensione dell'input

I risultati ottenuti necessitano di approfondimenti da eseguire in ulteriori sviluppi futuri. In particolare, occorre una analisi più approfondita alla ricerca delle motivazioni che spiegano questi risultati, cercando eventualmente soluzioni che li migliorino.

Conclusioni

In questa tesi abbiamo analizzato e implementato una prima versione semplificata dell'algoritmo che risolve il problema del matching per il Data Distribution Management. In particolare, abbiamo gettato le basi per realizzare una versione parallela dell'algoritmo SBM, che venga eseguita su GPU, per individuare le intersezioni all'interno di un solo insieme. Possiamo quindi concludere che è possibile implementare l'algoritmo su GPU. Sarà oggetto di sviluppi futuri portare a compimento quanto realizzato per ottenere un algoritmo che sia l'effettiva implementazione per GPU dell'algoritmo Sort Base Matching.

Tuttavia, abbiamo visto come la soluzione proposta presenti già dei limiti: la memoria e le prestazioni. Per quel che riguarda l'occupazione di memoria, sarà necessario individuare soluzioni che superino questo limite, cercando di capire se è necessario cambiare totalmente strategia oppure utilizzare ottimizzazioni che aiutino a risolvere i problemi. Ad esempio, si potrebbe tenere in considerazione che le matrici sono per natura molto sparse. Inoltre, capire se eventualmente è possibile memorizzare ogni cella non su un intero byte ma ridurre la dimensione fino al bit riducendo l'occupazione fino a 8 volte.

Infine, sarà necessario studiare in modo approfondito le prestazioni che ha questo algoritmo su GPU. In particolare, capire quali sono le criticità per ottimizzarlo e renderlo il più efficiente possibile. Questo è necessario affinché non rimanga solo un caso di studio, ma possa rappresentare una soluzione nei casi reali e rispondere alla richiesta di algoritmi sempre più veloci.

Bibliografia

[1] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Federate Interface Specification. *IEEE Std 1516.1-2010 (Revision of IEEE Std 1516.1-2000) - Redline*, pages 1–378, 2010.

[2] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Framework and Rules. *IEEE Std 1516-2010 (Revision of IEEE Std 1516-2000)*, pages 1–38, 2010.

[3] IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA)– Object Model Template (OMT) Specification. *IEEE Std 1516.2-2010 (Revision of IEEE Std 1516.2-2000)*, pages 1–110, 2010.

[4] Nathan Bell and Jared Hoberock. Chapter 26 - thrust: A productivity-oriented library for cuda. In Wen mei W. Hwu, editor, *GPU Computing Gems Jade Edition*, Applications of GPU Computing Series, pages 359–371. Morgan Kaufmann, Boston, 2012.

[5] G.E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[6] A. Boukerche, N.J. McGraw, C. Dzermajko, and Kaiyuan Lu. Grid-filtered region-based data distribution management in large-scale distributed simulation systems. In *38th Annual Simulation Symposium*, pages 259–266, 2005.

[7] Daniel Cederman and Philippas Tsigas. Gpu-quicksort: A practical quicksort algorithm for graphics processors. *ACM J. Exp. Algorithmics*, 14, January 2010.

[8] Nurhan Cetin, Kai Nagel, Bryan Raney, and Andreas Voellmy. Large-scale multi-agent transportation simulations. *Computer Physics Communications*, 147(1):559–564, 2002. Proceedings of the Europhysics Conference on Computational Physics Computational Modeling and Simulation of Complex Systems.

[9] Richard Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[10] Andrew Davidson, David Tarjan, Michael Garland, and John D. Owens. Efficient parallel merge sort for fixed and variable length keys. In *2012 Innovative Parallel Computing (InPar)*, pages 1–9, 2012.

[11] Erol Gelenbe and Fang-Jing Wu. Large scale simulation for human evacuation and rescue. *Computers & Mathematics with Applications*, 64(12):3869–3880, 2012. Theory and Practice of Stochastic Modeling.

[12] W. Daniel Hillis and Guy L. Steele. Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183, December 1986.

[13] Christian Jacob, Julius Litorco, and Leo Lee. Immunity through swarms: Agent-based simulations of the human immune system. In Giuseppe Nicosia, Vincenzo Cutello, Peter J. Bentley, and Jon Timmis, editors, *Artificial Immune Systems*, pages 400–412, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[14] Yu Jun, C. Raczy, and G. Tan. Evaluation of a sort-based matching algorithm for ddm. In *Proceedings 16th Workshop on Parallel and Distributed Simulation*, pages 62–69, 2002.

[15] Averill M. Law. How to build valid and credible simulation models. In *Proceedings of the 2009 Winter Simulation Conference (WSC)*, pages 24–33, 2009.

[16] Nikolaj Leischner, Vitaly Osipov, and Peter Sanders. Gpu sample sort. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–10, 2010.

[17] Moreno Marzolla and Gabriele D'Angelo. Parallel sort-based matching for data distribution management on shared-memory multiprocessors. In *2017 IEEE/ACM 21st International Symposium on Distributed Simulation and Real Time Applications (DS-RT)*, pages 1–8, 2017.

[18] Duane Merrill and Andrew Grimshaw. High performance and scalable radix sorting: A case study of implementing dynamic parallelism for gpu computing. *Parallel Processing Letters*, 21(02):245–272, 2011.

[19] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), 1965.

[20] C. Raczy, G. Tan, and J. Yu. A sort-based ddm matching algorithm for hla. *ACM Trans. Model. Comput. Simul.*, 15(1):14–38, January 2005.

[21] Erik Sintorn and Ulf Assarsson. Fast parallel gpu-sorting using a hybrid algorithm. *Journal of Parallel and Distributed Computing*, 68(10):1381–1388, 2008. General-Purpose Processing using Graphics Processing Units.

[22] G. Tan, Yusong Zhang, and R. Ayani. A hybrid approach to data distribution management. In *Proceedings Fourth IEEE International Workshop on Distributed Simulation and Real-Time Applications (DS-RT 2000)*, pages 55–61, 2000.

[23] M. Wheat and D.J. Evans. An efficient parallel sorting algorithm for shared memory multiprocessors. *Parallel Computing*, 18(1):91–102, 1992.