Università degli Studi di Padova

Dipartimento di Ingegneria Industriale
Corso di Laurea in Ingegneria Aerospaziale

Sviluppo di un modulo software per la simulazione su GPU della dinamica del plasma per propulsione spaziale elettrica

Laureando:
Alberto Madonna

Relatore:
Ch.mo Prof. Daniele Pavarin
Correlator:
Dr. Marco Manente

Anno accademico 2012/2013

Ai miei genitori,
che hanno sperato ed atteso questi momenti più di chiunque altro

Indice

Elenco delle figure

Sommario

Questa tesi affronta la sintesi teorica e la realizzazione pratica di moduli software che implementino una simulazione della dinamica del plasma su un processore massivamente parallelo come la GPU, attraverso il modello di programmazione CUDA. Il nostro punto di partenza è il codice F3MPIC, basato sull'algoritmo Particle-in-Cell e sviluppato dal CISAS come strumento numerico per la ricerca sui propulsori al plasma. Per superare le difficoltà poste dalle caratteristiche di F3MPIC ad una efficace esecuzione su GPU, integriamo soluzioni provenienti dallo stato dell'arte nella simulazione parallela del plasma, e aggiungiamo alcuni contributi originali, primo fra tutti un innovativo uso di un partizionamento adattivo del dominio di calcolo. Ne scaturisce un nuovo algoritmo di simulazione su GPU, pensato per complementarsi naturalmente con il codice originale, ma con la possibilità dimostrata di un sostanzioso aumento di prestazioni. Un confronto netto tra le funzionalità adattate al nuovo paradigma di programmazione rivela tempi di calcolo oltre 8 volte inferiori. Nel complesso, il lavoro svolto (il primo a confrontarsi con molte nuove problematiche), le indicazioni prestazionali e le opportunità offerte dal nuovo software qualificano questa esperienza come un punto di partenza che apre a numerosi sviluppi, anche di carattere multidisciplinare.

Introduzione

Le varie tecnologie di propulsione elettrica ad uso spaziale si pongono come soluzioni competitive per il controllo d'assetto o la propulsione di spacecraft in missioni di lunga durata, grazie alle loro caratteristiche di alto impulso specifico ponderale e basso consumo di propellente, nonostante valori assoluti di spinta molto bassi.

Il CISAS (Centro Interdipartimentale per gli Studi e Attività Spaziali) dell'Università di Padova è attivo nella ricerca di nuove soluzioni tecnologiche per la propulsione elettrica al plasma, e ha coordinato il progetto europeo di ricerca HPH.com, con la finalità di progettare e collaudare un mini propulsore con sorgente helicon per operazioni di controllo d'assetto e posizione di satelliti.

All'interno di questo progetto, per coadiuvare le attività sperimentali con strumenti di ottimizzazione ed indagine numerica, il CISAS ha sviluppato un pacchetto software per la simulazione della dinamica del plasma denominato F3MPIC e basato sull'algoritmo di simulazione Particle-in-cell (PIC). Questo algoritmo mira a riprodurre il comportamento del plasma attraverso la reciproca interazione tra un'insieme di macro-particelle lagrangiane (rappresentanti le specie fisiche elettricamente cariche) ed una griglia spaziale fissa (che riproduce i campi elettromagnetici presenti nello spazio) all'interno della quale le particelle si muovono: la quantità di moto delle particelle viene influenzata dai campi elettromagnetici per mezzo della forza di Lorentz, ma essendo cariche le particelle spostandosi inducono variazioni dei campi stessi. L'algoritmo PIC, nella sua forma più semplice, si compone di quattro stadi (interpolazione, integrazione del moto delle particelle, deposizione, integrazione dei campi) che vengono ripetuti ciclicamente per ricostruire l'evoluzione del plasma nel periodo di tempo stabilito per la simulazione; è possibile introdurre nel flusso della simulazione altri stadi di calcolo per riprodurre fenomeni fisici più complessi o specifici.

Questa metodologia di simulazione è nota per richiedere notevoli risorse al fine di un esito significativo, ed i tempi di calcolo sono spesso molto lunghi. Tuttavia, l'algoritmo PIC ha le caratteristiche per prestarsi ad un'implementazione su un processore di calcolo massivamente parallelo come l'Unità di Elaborazione Grafica (GPU).

Infatti, la competizione nel campo della computer grafica in tempo reale negli ultimi anni ha trasformato questi dispositivi in processori paralleli dalle elevate potenzialità di calcolo. L'introduzione da parte di NVIDIA del modello di programmazione CUDA ha contribuito a permettere una larga adozione di queste tecnologie nel campo del calcolo tecnico-scientifico. CUDA espone l'hardware della GPU in modo trasparente, servendosi alle stesso tempo di pratiche astrazioni per esprimere in maniera semplice e scalabile il parallelismo del problema da risolvere. Per queste ed altre ragioni, legate alla prestazionalità e disponibilità delle risorse, CUDA è stato preferito a soluzioni concorrenti.

Scopo del lavoro presentato in questa tesi è stato quello di realizzare dei componenti software che inserendosi in F3MPIC, permettano di accelerare le simulazioni tramite l'uso della GPU, mantenendo il più possibile le funzionalità del codice originale ed una semplicità di utilizzo per un generico utilizzatore finale. Rispettare i paradigmi di F3MPIC (sopratutto il suo utilizzo di

griglie non strutturate) ci pone tuttavia di fronte a delle problematiche scarsamente affrontate prima d'ora. Il primo passo è stato condurre una meticolosa ricerca in letteratura riguardante la simulazione PIC su GPU. È stato poi necessario introdurre dei contributi originali per sintetizzare un nuovo algoritmo di simulazione adatto alla GPU.

Il resto del documento è strutturato come segue. Nel primo capitolo si introducono brevemente le tecnologie di propulsione elettrica per lo spazio. Nel secondo capitolo si descrivono i concetti teorici di fisica dei plasmi utili ai fini del presente lavoro. Il terzo capitolo è dedicato ad un'introduzione generale al metodo PIC per la simulazione del plasma e all'approfondimento delle sue fasi. Il quarto capitolo propone un'introduzione al modello di programmazione CUDA. Il quinto capitolo espone i risultati della nostra indagine sullo stato dell'arte della simulazione PIC su GPU. Il sesto capitolo tratta la sintesi di una nuova procedura di calcolo incentrata sulla simulazione propulsiva tramite F3MPIC. Il settimo capitolo affronta il collaudo del codice realizzato, con il commento e l'analisi dei risultati.

Parte I

Fondamenti teorici

Capitolo 1

La propulsione spaziale elettrica

Si definisce sistema di propulsione elettrico ad uso spaziale qualsiasi sistema utilizzi energia elettrica per alterare la velocità di un veicolo spaziale.

La propulsione elettrica è generalmente fondata sull'accelerazione ed espulsione di particelle elettricamente cariche, impiegate come massa di reazione per accelerare il veicolo. La velocità di espulsione di queste particelle non ha nessun limite intrinseco (se non la velocità della luce), ma dipende solo dalla potenza disponibile e dalla spinta richiesta, a differenza di quanto avviene nei propulsori chimici, dipendenti dalle caratteristiche dei reagenti e dalla temperatura.

La propulsione elettrica mira ad ottenere spinta con elevate velocità di scarico, che risulta in una riduzione della quantità di propellente richiesto per una data missione o applicazione rispetto ad altri metodi di propulsione convenzionale. Una ridotta massa di propellente può significativamente diminuire la massa al lancio di uno spacecraft o satellite, portando all'uso di vettori di lancio più piccoli e a costi inferiori. Per contro, i motori elettrici richiedono ovviamente un sistema di produzione di potenza elettrica a bordo, e richiedono attenzione per il controllo termico e l'alto peso dei sistemi ausiliari.

La combinazione di questi aspetti fornisce le caratteristiche peculiari dei propulsori elettrici: una spinta molto ridotta ($< 0.1N$), ma elevatissimo impulso specifico ($> 1000s$) e lunga durata. Applicazioni comuni per questi propulsori sono il mantenimento dell'assetto, lo station-keeping di satelliti, o il ruolo di propulsore principale per uno spacecraft in missioni di lunga durata verso lo spazio profondo.

Ricaviamo le equazioni di base della propulsione elettrica. La spinta è ovviamente data dalla variazione temporale della quantità di moto, che può essere scritta come

$$T = \frac{d}{dt}\left(m_p v_{ex}\right) = \dot{m}_p v_{ex}, \tag{1.1}$$

dove $\dot{m}_p$ è la portata di massa di propellente in kg/s e v_{ex} è la velocità di scarico.

La potenza in ingresso è

$$P = \dot{m}_p \frac{v_{ex}^2}{2\eta}, \tag{1.2}$$

con η l'efficienza di conversione, mentre la potenza di spinta cinetica del getto, chiamata *jet power*, è definita come

$$P_{jet} = \frac{1}{2}\dot{m}_p v_{ex}^2 = \frac{T^2}{2\dot{m}_p} = \eta P. \tag{1.3}$$

L'impulso specifico a spinta e portata di massa costante è

$$I_{sp} = \frac{T}{\dot{m}_p g}. \tag{1.4}$$

L'equazione della spinta per la propulsione elettrica può quindi essere scritta come

$$T = \frac{2\eta P}{g_0 I_{sp}}. \tag{1.5}$$

I motori elettrici si distinguono generalmente in base alla metodologia di accelerazione usata per ottenere la spinta. Questi metodi si possono facilmente separare in tre categorie:

- **Elettrotermici**: campi elettromagnetici vengono usati per scaldare il propellente e poi espellerlo attraverso un ugello. La ionizzazione del propellente è scarsa o nulla. Appartengono a questa categoria i resistojet e gli arcjet.
- **Elettrostatici**: l'accelerazione è dovuta principalmente alla forza di Coulomb, ovvero all'applicazione di un campo elettrostatico nella direzione dell' accelerazione. Ne fanno parte i motori a ioni e quelli colloidali.
- **Elettromagnetici**: l'accelerazione è causata dalla combinazione di un campo elettrostatico e da un campo magnetico. Esempi di motori elettromagnetici sono i propulsori ad effetto Hall, i Pulsed Plasma Thruster e i motori Helicon.

I motori elettrostatici ed elettromagnetici operano su particelle cariche prodotte dalla ionizzazione di un gas propellente, che crea ioni ed elettroni, formando quello che viene chiamato un plasma. Il plasma per applicazioni spaziali costituisce un propellente di facile conservazione, dà la possibilità di modulare la spinta e può essere considerato globalmente neutro. Un sistema di propulsione al plasma normalmente è composto da 3 stadi: produzione (tramite sorgente capacitiva-induttiva o sorgente wave), accelerazione e distacco (servendosi di un ugello magnetico o di un neutralizzatore).

Chiarito il ruolo del plasma nella propulsione spaziale, per gli scopi del presente lavoro concentriamo ora la nostra attenzione sulle caratteristiche del plasma come entità fisica e sulla sua simulazione.

Capitolo 2

Basi della fisica dei plasmi

Un plasma è un gas *quasi-neutro* di particelle cariche e neutre che generano campi elettromagnetici e interagiscono con essi, mostrando un comportamento collettivo.

La condizione di quasi-neutralità si può esprimere come

$$-q_e n_e = q_i n_i \pm \Delta \quad con\ \Delta\ piccolo, \tag{2.1}$$

dove q_e e q_i sono rispettivamente le cariche di un elettrone e di uno ione, ed n_e e n_i sono le rispettive densità volumetriche. La condizione di comportamento collettivo poggia invece sull'ipotesi che le interazioni tra particelle siano trascurabili rispetto agli effetti collettivi; ciò è possibile grazie ad effetti come la schermatura di Debye (*Debye shielding*) ed alla condizione che la sezione d'urto delle collisioni tra elettroni e particelle neutre sia molto minore di quella delle collisioni tra ioni e neutri: in questo modo la dinamica del plasma è determinata dalle leggi elettromagnetiche e non da quelle idrodinamiche. Analizzeremo più rigorosamente queste ipotesi in seguito.

In principio, la dinamica di un plasma è completamente determinata considerando che la forza agente su ogni particella relativistica è la forza di Lorentz e l'evoluzione dei campi elettromagnetici è governata dalle equazioni di Maxwell. In CGS ($\mathbf{x}_i$, $\mathbf{p}_i = m_i \gamma_i \mathbf{v}_i$ sono la posizione ed il momento lineare della i-esima particella):

$$\begin{cases} \dot{\mathbf{x}}_i &= \mathbf{v}_i \\ \dot{\mathbf{p}}_i &= q_i \left(\mathbf{E}(\mathbf{x}_i) + \dfrac{\mathbf{v}_i \times \mathbf{B}(\mathbf{x}_i)}{c} \right) \end{cases} \tag{2.2}$$

$$\begin{cases} \nabla \cdot \mathbf{B} &= 0 \\ \nabla \cdot \mathbf{E} &= 4\pi\rho \\ \nabla \times \mathbf{B} - \frac{1}{c}\dfrac{\partial \mathbf{E}}{\partial t} &= \frac{4\pi}{c}\mathbf{j} \\ \nabla \times \mathbf{B} + \dfrac{1}{c}\dfrac{\partial \mathbf{B}}{\partial t} &= 0 \end{cases} \tag{2.3}$$

Dalla seconda delle equazioni di Maxwell (2.3) nel caso elettrostatico, possiamo ricavare

$$\nabla \times \mathbf{E} = 0. \tag{2.4}$$

Dal momento che il rotore di un gradiente è nulla, la precedente significa che il campo elettrico può essere derivato dal gradiente di un potenziale scalare Φ:

$$\mathbf{E} = -\nabla\Phi; \tag{2.5}$$

se introdotto nella prima equazione di Maxwell, questo risultato ci permette di ottenere l'utilissima *Equazione di Poisson*:

$$\nabla^2\Phi = -\frac{\rho}{\varepsilon_0}. \tag{2.6}$$

Nelle ultime delle (2.3), le quantità ρ e $\mathbf{j}$ sono calcolate a partire dalla distribuzione delle particelle nello spazio fase senza ricorrere ad alcuna operazione di media spaziale: questo garantisce di includere nel modello le collisioni binarie.

Questo approccio è molto poco pratico, sia per quanto concerne simulazioni numeriche (il numero di particelle da considerare non è realisticamente computabile), sia per la teoria analitica (le equazioni non lineari alle derivate parziali che descrivono i fenomeni non sono risolvibili analiticamente). Tuttavia, una varietà di considerazioni fisiche ed approssimazioni possono portare ad altrettanti modelli semplificati.

2.1 Distribuzione di velocità e temperatura

Un gas in equilibrio termico ha particelle con velocità molto differenti. La distribuzione più probabile è nota come distribuzione *Maxwelliana*, che monodimensionalmente è data da

$$f(u) = A\, exp\left(\frac{-mu^2}{2k_BT}\right) \tag{2.7}$$

dove con k_B si indica la costante di Boltzmann ($1.38 \cdot 10^{-23} J/K$). La densità è data da

$$n = \int_{-\infty}^{\infty} f(u). \tag{2.8}$$

La larghezza della distribuzione è caratterizzata da T, detta Temperatura, mentre l'altezza è legata alla densità da $A = n\left(\frac{m}{2\pi k_BT}\right)^{\frac{1}{2}}$.

Se si calcola l'energia cinetica media $\mathscr{E}_{av}$ delle particelle per una distribuzione monodimensionale si ottiene:

$$\mathscr{E}_{av} = \frac{1}{2}k_BT. \tag{2.9}$$

Dato lo stretto rapporto tra energia e temperatura si usa indicare la temperatura del plasma in unità di energia:

$$1eV = 1.6 \cdot 10^{-19} J = 11600K. \tag{2.10}$$

2.2 Frequenza di plasma e frequenza di ciclotrone

Consideriamo un plasma neutro (ioni con tasso di ionizzazione $Z = 1$), monodimensionale, inizialmente uniforme, non relativistico e con ioni sufficientemente massivi da poter essere considerati come fissi nella scala temporale del nostro processo. Consideriamo inoltre solo campi elettrostatici e trascuriamo effetti termici.

In queste ipotesi la forza elettrostatica $F = qE$ si può riscrivere come

$$m_e\frac{d^2x}{dt_2} = -\frac{q_e\rho x}{\varepsilon_0} \Rightarrow \frac{d^2x}{dt_2} = -\omega_{pe}^2 x \tag{2.11}$$

che è la frequenza di un oscillatore armonico con frequenza

$$\omega_{pe} = \sqrt{\frac{q_e \rho}{m_e \varepsilon_0}} = \sqrt{\frac{q_e^2 n_e}{m_e \varepsilon_0}}, \tag{2.12}$$

detta *frequenza elettronica di plasma*.

Si consideri ora una singola particella carica, sia essa uno ione o un elettrone, situata in una regione dello spazio con campo magnetico $\mathbf{B}$ uniforme e campo elettrico $\mathbf{E} = 0$.

In questo caso la particella presenta un moto di girazione di ciclotrone

$$m\frac{d\mathbf{v}}{dt} = q\mathbf{v} \times \mathbf{B}. \tag{2.13}$$

Nel caso che $\mathbf{B} = B\hat{\mathbf{z}}$, possiamo esplicitare la precedente equazione vettoriale come

$$\begin{bmatrix} m\dot{v}_x \\ m\dot{v}_y \\ m\dot{v}_z \end{bmatrix} = q \begin{vmatrix} \hat{x} & \hat{y} & \hat{z} \\ v_z & v_y & v_z \\ 0 & 0 & B \end{vmatrix} = \begin{bmatrix} qBv_y \\ -qBv_x \\ 0 \end{bmatrix}, \tag{2.14}$$

che derivate diventano

$$\ddot{v}_x = \frac{qB}{m}\dot{v}_y = -\left(\frac{qB}{m}\right)^2 v_x \qquad \ddot{v}_y = -\frac{qB}{m}\dot{v}_x = -\left(\frac{qB}{m}\right)^2 v_y; \tag{2.15}$$

queste equazioni descrivono un oscillatore armonico con frequenza

$$\omega_c = \frac{|q|B}{m}, \tag{2.16}$$

detta *frequenza di ciclotrone*, che per la convenzione scelta è sempre non negativa.

La soluzione delle equazioni (2.15) è quindi

$$v_{x,y} = v_\perp \, exp(\pm i\omega_c t + i\delta_{x,y}) \tag{2.17}$$

con *pm* che indica il segno della carica. La costante positiva di integrazione $v_\perp$ è la velocità perpendicolare al campo $\mathbf{B}$: quindi la particella ha un moto giratorio con raggio

$$r_L = \frac{v_\perp}{\omega_c} = \frac{mv_\perp}{|q|B}, \tag{2.18}$$

detto *raggio di Larmor*. Integrando le (2.17) si ottengono le espressioni

$$x - x_0 = r_L \sin\omega_c t \qquad y - y0 = r_L \cos\omega_c t, \tag{2.19}$$

che descrivono un'orbita circolare attorno ad un centro guida che è fisso.

2.3 Lunghezza di Debye e parametro di plasma

Si consideri un plasma completamente ionizzato, e si indichino ancora con n_e ed n_i le densità di elettroni e di ioni. Il plasma sia vicino all'equilibrio termico alla temperatura T e, allo stato indisturbato, si consideri $n_e = n_i = n_0$ in ogni punto dello spazio.

Se perturbiamo il sistema introducendo una carica positiva puntiforme Q, si osserva che essa attirerà gli elettroni e respingerà gli ioni, creando una nuvola di cariche negative che scherma il campo e potenziale elettrici della carica puntiforme. Gli elettroni, inoltre, non possono collassare sulla carica puntiforme (e così facendo neutralizzarla) a causa della loro stessa energia termica.

E' possibile stimare l'estensione di questo effetto schermante. La distribuzione Maxwelliana degli elettroni in presenza di un potenziale esterno assume la forma

$$f(u) = A\, exp\left(\frac{-\frac{1}{2}mu^2 - q_e\Phi}{k_B T_e}\right). \tag{2.20}$$

Integrando su u, sapendo che per $\Phi \to 0$ si ha $n_i = n_e = n_\infty$, si ottiene la *relazione di Boltzmann*

$$n_e = n_\infty\, exp\left(\frac{q_e\Phi}{k_B T_e}\right). \tag{2.21}$$

Monodimensionalmente l'equazione di Poisson (2.6) diviene, con ionizzazione $Z = 1$

$$\varepsilon_0 \nabla^2\Phi = \varepsilon_0 \frac{d^2\Phi}{dx^2} = -q_e(n_i - n_e); \tag{2.22}$$

usando (2.21) si ottiene

$$\varepsilon_0 \frac{d^2\Phi}{dx^2} = en_\infty \left[exp\left(\frac{q_e\Phi}{k_B T_e}\right) - 1\right]. \tag{2.23}$$

Nell'intervallo in cui $|\frac{q_e\Phi}{k_B T_e}| \ll 1$ è possibile espandere in serie di Taylor:

$$\varepsilon_0 \frac{d^2\Phi}{dx^2} = en_\infty \left[\frac{q_e\Phi}{k_B T_e} + \frac{1}{2}\left(\frac{q_e\Phi}{k_B T_e}\right)^2 + \dots\right]. \tag{2.24}$$

Arrestando la serie al primo grado si trova che

$$\varepsilon_0 \frac{d^2\Phi}{dx^2} = \frac{n_i n f t y q_e^2}{k_B T_e}\Phi, \tag{2.25}$$

quindi la soluzione è nella forma

$$\Phi = \Phi_0\, exp\left(\frac{-|x|}{\lambda_D}\right), \tag{2.26}$$

dove λ_D è detta *lunghezza di Debye*:

$$\lambda_D = \sqrt{\frac{\varepsilon_0 k_B T_e}{n_e q_E^2}}. \tag{2.27}$$

Si può interpretare la lunghezza di Debye come la scala spaziale nella quale il plasma scherma il potenziale elettrostatico generato da cariche puntiformi.

Il numero di particelle in una sfera di Debye è chiamato *parametro di plasma* Λ:

$$\Lambda = \frac{4}{3} n \lambda_D^3. \tag{2.28}$$

Ricordando la definizione di frequenza di plasma (2.12) e introducendo la *velocità termica dell'elettrone*

$$v_{th} = \sqrt{\frac{kT}{m_e}} \tag{2.29}$$

possiamo scrivere

$$\lambda_D = \frac{v_{th}}{\omega_{pe}} = \frac{2\pi}{T_{pe}}. \tag{2.30}$$

La lunghezza di Debye, dunque, corrisponde anche alla lunghezza percorsa da un'elettrone in un intervallo di tempo $\frac{T_{pe}}{2\pi}$, ed è inoltre la scala spaziale su cui ogni sbilanciamento viene neutralizzato, poiché nella scala temporale di T_{pe} il plasma mantiene la propria neutralità. Pertanto, se la lunghezza del sistema L è molto maggiore di λ_D, il plasma può essere considerato neutro. Questa schermatura è statisticamente significativa si all'interno di λ_D si trova un adeguato numero di particelle cariche, che si può indicare con la seguente condizione sul parametro di plasma:

$$\Lambda = \frac{4}{3} n \lambda_D^3 \ggg 1. \tag{2.31}$$

2.4 Modello fluido e modello cinetico

L'equazione del moto di una singola particella è

$$m\frac{d\mathbf{v}}{dt} = q(\mathbf{E} + \mathbf{v} \times \mathbf{B}). \tag{2.32}$$

Le equazioni del plasma come un *n-fluido* (ogni specie j è trattata come un fluido separato ed interagente con gli altri per mezzo dei campi elettromagnetici) si possono ricavare semplicemente moltiplicando per la densità della rispettiva specie

$$mn\frac{d\mathbf{v}}{dt} = qn(\mathbf{E} + \mathbf{v} \times \mathbf{B}). \tag{2.33}$$

Dal momento che data una funzione $\mathbf{G}(x,t)$ si ha che

$$\frac{d\mathbf{G}}{dt} = \frac{\partial \mathbf{G}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{G}, \tag{2.34}$$

allora con $\mathbf{G} = \mathbf{v}$ si ottiene l'equazione del fluido con derivata convettiva

$$mn\left[\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v}\right] = qn(\mathbf{E}\mathbf{v} \times \mathbf{B}). \tag{2.35}$$

In un fluido, la direzione del moto e le componenti della quantità di moto sono specificate dal tensore di stress $\mathbf{P}$, di componenti $P_{ij} = mnv_i\bar{v}_j$. Nel caso di una distribuzione di particelle isotropa allora

$$\mathbf{P} = \begin{bmatrix} p & 0 & 0 \\ 0 & p & 0 \\ 0 & 0 & p \end{bmatrix} = p \tag{2.36}$$

quindi la (2.35) si modifica, tenendo conto della forza dovuta al tensore di stress (o del semplice gradiente di pressione), in

$$mn\left[\frac{\partial \mathbf{v}}{\partial t} + (\mathbf{v} \cdot \nabla)\mathbf{v}\right] = qn(\mathbf{E}\mathbf{v} \times \mathbf{B}) - \nabla\mathbf{P}. \tag{2.37}$$

Introduciamo ora l'equazione di continuità per la densità delle particelle

$$\frac{\partial n}{\partial t} + \nabla \cdot (n\mathbf{u}) = 0 \tag{2.38}$$

e l'equazione di stato termodinamico

$$p = Cn^\gamma \tag{2.39}$$

con γ come rapporto tra i calori specifici e quindi

$$\frac{\nabla p}{p} = \gamma \frac{\nabla n}{n} \tag{2.40}$$

e per una isoterma (conducibilità termica infinita) si ha

$$\nabla p = \nabla(n k_B T) = k_B T \nabla n \tag{2.41}$$

dove è stato assunto $\gamma = 1$.

Definendo $\rho = n_i q_i + n_e q_e$, $\mathbf{j} = n_i q_i \mathbf{u}_i + n_e q_e \mathbf{u}_e$ e ignorando sia le collisioni (comprese nell'interpretazione fluida) sia la viscosità (il tensore di stress coincide con la pressione scalare p), si ha per ciascuna specie

$$\begin{array}{rcl} \dfrac{\partial n_j}{\partial t} + \nabla \cdot (n_j \mathbf{u}_j) & = & 0 \\ m_j n_j \left[\dfrac{\partial \mathbf{v}_j}{\partial t} + (\mathbf{v}_j \cdot \nabla)\mathbf{v}_j \right] & = & q_j n_j (\mathbf{E}\mathbf{v}_j \times \mathbf{B}) - \nabla \mathbf{p}_j \quad con \quad p_j = C_j n_j^{\gamma_j}. \end{array} \tag{2.42}$$

I campi elettromagnetici sono dati dalle Equazioni di Maxwell, che chiudono il sistema.

$$\left\{ \begin{array}{rcl} \nabla \cdot \mathbf{B} & = & 0 \\ \nabla \cdot \mathbf{E} & = & 4\pi\rho \\ \nabla \times \mathbf{B} & = & \mu_0 \left(\mathbf{j} + \frac{1}{c} \dfrac{\partial \mathbf{E}}{\partial t} \right) \\ \nabla \times \mathbf{E} & = & -\dfrac{\partial \mathbf{B}}{\partial t} \end{array} \right. \tag{2.43}$$

Il modello cinetico di un plasma descrive invece lo stato delle particelle del sistema attraverso una funzione di distribuzione $f_j(\mathbf{x}, \mathbf{p}, t)$, che rappresenta la densità delle particelle (per la specie j) nello spazio delle fasi ($\mathbf{x}, \mathbf{p} = \gamma m \mathbf{v}$), in modo che

$$f_j(\mathbf{x}, \mathbf{p}, t) d\mathbf{x} d\mathbf{p} = dN(\mathbf{x}, \mathbf{p}, t) \tag{2.44}$$

sia il numero di particelle nell'elemento di volume $dV = d\mathbf{x}d\mathbf{p}$ dello spazio delle fasi. L'elemento $d\mathbf{x}$ non deve essere tanto piccolo da non contenere un numero statisticamente significativo di particelle. Ciò consente di approssimare la $f_j(\mathbf{x}, \mathbf{p}, t)$ (che è funzione di 7 variabili) come una funzione continua. Le quantità macroscopiche (osservabili) sono ottenute mediando sulle quantità di moto:

$$\begin{array}{rcll} n_j(\mathbf{x}) & = & \int f_j(\mathbf{x}, \mathbf{p}, t) d\mathbf{p} & \textit{densità particelle} \\ n_j \mathbf{u}_j(\mathbf{x}) & = & \int \mathbf{v} f_j(\mathbf{x}, \mathbf{p}, t) d\mathbf{p} & \textit{velocità media} \\ \left[P^{kl} \mathbf{x} \right]_j & = & m_j \int \mathbf{v}_k \mathbf{v}_l f_j(\mathbf{x}, \mathbf{p}, t) d\mathbf{p} & \textit{pressione media} \\ & & & \\ \rho_j \mathbf{x} & = & q_j \int f_j(\mathbf{x}, \mathbf{p}, t) d\mathbf{p} & \textit{densità di carica} \\ \mathbf{j}_j(\mathbf{v}) & = & q_j \int \mathbf{v} f_j(\mathbf{x}, \mathbf{p}, t) d\mathbf{p} & \textit{densità di corrente} \end{array} \tag{2.45}$$

La funzione di distribuzione cambierà per effetto del flusso di particelle attraverso la superficie dell'elemento $d\mathbf{x}d\mathbf{p}$, sia per la velocità delle particelle, sia per effetto delle collisioni. Per esigenze di conservazione il flusso di particelle attraverso la superficie S che delimita l'elemento ΔV (esadimensionale) più quelle generate dalle collisioni deve eguagliare il rateo temporale di variazione della densità delle particelle nello spazio delle fasi. Applicando il teorema della

divergenza, tenendo conto dell'indipendenza di $\mathbf{x}$ e $\mathbf{p}$ e semplificando si giunge all'*equazione di Boltzmann*:

$$\frac{\partial f_j}{\partial t} + \mathbf{v} \cdot \frac{\partial f_j}{\partial \mathbf{x}} + q_j \left(\mathbf{E} + \frac{\mathbf{v} \times \mathbf{B}}{c} \right) \cdot \frac{\partial f_j}{\partial \mathbf{p}} = \left(\frac{\partial f}{\partial t} \right)_{collision}. \tag{2.46}$$

Il termine a primo membro è la derivata totale (o anche derivata convettiva) della funzione di distribuzione, dunque l'equazione di Boltzmann è un'equazione di continuità esadimensionale per f. Su una scala temporale molto più piccola delle frequenze di collisione, l'equazione afferma che la derivata convettiva è nulla e viene detta *equazione di Vlasov*:

$$\frac{\partial f_j}{\partial t} + \mathbf{v} \cdot \frac{\partial f_j}{\partial \mathbf{x}} + q_j \left(\mathbf{E} + \frac{\mathbf{v} \times \mathbf{B}}{c} \right) \cdot \frac{\partial f_j}{\partial \mathbf{p}} = 0. \tag{2.47}$$

Il modello viene chiuso accoppiando la precedente espressione con le equazioni del moto della singola particella (2.2) e con le equazioni di Maxwell (2.3).

Si fa notare che è possibile ottenere il modello fluido da quello cinetico considerando i momenti dell'equazione di Vlasov. I primi due sono:

$$\int d\mathbf{p} \left[\frac{\partial f_j}{\partial t} + \mathbf{v} \cdot \frac{\partial f_j}{\partial \mathbf{x}} + q_j \left(\mathbf{E} + \frac{\mathbf{v} \times \mathbf{B}}{c} \right) \cdot \frac{\partial f_j}{\partial \mathbf{p}} \right] = 0 \tag{2.48}$$

$$\int d\mathbf{p}\mathbf{p} \left[\frac{\partial f_j}{\partial t} + \mathbf{v} \cdot \frac{\partial f_j}{\partial \mathbf{x}} + q_j \left(\mathbf{E} + \frac{\mathbf{v} \times \mathbf{B}}{c} \right) \cdot \frac{\partial f_j}{\partial \mathbf{p}} \right] = 0 \tag{2.49}$$

Ricaviamo così nuovamente le equazioni di continuità per la densità particellare e l'equazione di moto del fluido. Le precedenti integrazioni sono eseguite considerando che un momento lineare del secondo ordine porterebbe all'equazione del tensore di pressione P, il quale altro non è che il tensore di stress isotropo. Considerazioni legate al rapporto tra lunghezza d'onda caratteristica, frequenza del sistema e la velocità termica delle particelle della j-esima specie ci portano infine a definire un'equazione di stato termodinamica.

Le rappresentazioni fluida e cinetica sono dunque equivalenti e riflettono la natura del plasma, che talora si comporta come un fluido, talora come un insieme di particelle singole. La scelta del modello matematico più opportuno per una data occasione viene fatta considerando la densità di particelle cariche e la frequenza delle collisioni. Se prevale il comportamento collettivo l'interpretazione fluida è appropriata, mentre in fenomeni a bassa densità o dalle caratteristiche più raffinate si rende necessaria l'interpretazione cinetica.

Capitolo 3

Simulare il plasma

Nell'approcciare la simulazione di plasmi neutri, uno dei primi elementi da considerare è la lunghezza di Debye, in particolare riguardo al fatto che i sistemi di interesse hanno grandezze caratteristiche $L \gg \lambda_D$. In questa prospettiva, si consideri che i plasmi di vapori alcalini hanno un parametro di plasma $\Lambda \approx 10^2$, la ionosfera ha $\Lambda \approx 10^4$ e plasmi confinati magneticamente in esperimenti di fusione hanno $\Lambda \approx 10^6$. Inoltre, in un modello tridimensionale, lo spazio delle fasi del plasma è esadimensionale, pertanto la memoria necessaria per simulazione scala con n^6, dove n è il numero di particelle simulate. Ci si accorge ben presto che una simulazione diretta di questi sistemi è ben oltre gli attuali limiti tecnologici, sia in termini di prestazioni di calcolo che di quantitativi di memoria.

E' dunque necessario usare una rappresentazione approssimata della distribuzione $f_j(\mathbf{x}, \mathbf{p}, t)$. Il metodo di maggiore impiego, è il cosiddetto *Particle-in-cell (PIC)*.

Esso decompone la f_j nella somma di contributi provenienti da un insieme finito di N_{p_j} macro-particelle o *quasi-particelle*. Le traiettorie di queste particelle virtuali vengono seguite nello spazio delle fasi da un punto di vista lagrangiano, mentre i campi elettromagnetici sono discretizzati su una griglia spaziale. Il metodo particle-in-cell si pone pertanto come una formulazione ibrida Lagrangiana - Euleriana.

Le macro-particelle non sono cariche puntiformi, ma sono caratterizzate da una dimensione finita e da una funzione di densità che si estende nello spazio, così da poter essere considerate come nuvole di carica continue e ridurre il rumore dovuto all'approssimazione numerica.

Tali semplificazioni sono giustificate dal fatto che, in primo luogo, si è interessati al comportamento collettivo di plasmi senza collisioni a lunghezze d'onda maggiori della lunghezza di Debye; in secondo luogo, è possibile ottenere informazioni molto utili sul comportamento essenziale del plasma senza soddisfare rigorosamente il requisito tridimensionale di, per esempio, $\Lambda \approx 10^6$.

Approfondiamo brevemente queste ultime affermazioni. Il comportamento fisico di un plasma è dato da elettroni e ioni che si muovono nei loro campi di Coulomb con sufficiente energia cinetica per impedire la ricombinazione. Dunque, un'altra caratterizzazione del plasma è

$$\frac{Energia\ Cinetica\ Termica\ (KE)}{Energia\ Potenziale\ Microscopica\ (PE)} \gg 1 \tag{3.1}$$

Questo rapporto, per plasmi da laboratorio, è in effetti Λ. Tuttavia, la fisica fondamentale richiede solo che $KE \gg PE$, che può essere soddisfatto da valori di Λ bassi come 10, o può essere ottenuto con mezzi differenti dall'imporre $\Lambda \gg 1$, se necessario. Anche la caratterizzazione di assenza di collisioni può essere accettabile con $\Lambda \approx 10$.

Possiamo dunque concludere che le simulazioni particellari comportano delle approssimazioni più che accettabili per lo studio dei plasmi. Le simulazioni basate su particelle si so-

no anche rivelate più flessibili rispetto all'integrazione dell'equazione cinetica non collisionale (chiamata equazione di Maxwell-Vlasov), soprattutto in problemi dimensionali e in un'accurata rappresentazione dello spazio delle velocità su lunghi tempi di simulazione.

3.1 Il metodo Particle-in-cell

Il metodo PIC è caratterizzato, come già accennato, da un accoppiamento bidirezionale tra una griglia spaziale ed un insieme di particelle lagrangiane e prevede una serie di stadi di esecuzione da ripetere ciclicamente per un numero finito di istanti temporali.

Gli stadi sono i seguenti:

1. **Avanzamento delle particelle:** Le quantità particellari, come velocità e posizione, sono note alle particelle e possono assumere tutti i valori nello spazio x e v, chiamato *spazio delle fasi*. Attraverso la conoscenza del campo elettromagnetico (che all'istante $t = 0$ viene da appropriate condizioni iniziali), le forze sulle singole particelle vengono calcolate usando l'equazione di Lorentz, ed a loro volta impiegate nell'equazione del moto di Newton.

2. **Deposizione:** Si ricavano i valori delle densità di corrente e di carica ai nodi della griglia con le nuove posizioni e velocità delle particelle. Questo processo di assegnazione di cariche e correnti implica un'operazione di *pesatura* dei nodi che dipende dalle posizioni delle singole particelle.

3. **Calcolo dei campi elettrici e magnetici:** I campi vengono integrati attraverso le equazioni di Maxwell e le densità calcolate nello stadio precedente

4. **Interpolazione:** I valori dei nuovi campi **E** e **B** alle posizioni delle singole particelle vengono ricavati con un'interpolazione dai nodi della griglia, che fa uso degli stessi pesi impiegati nella deposizione per ragioni di conservazione delle quantità fisiche.

A seconda delle capacità delle quali si vuole dotare il codice di simulazione, si può introdurre un ulteriore stadio tra l'integrazione e la deposizione, che si occupi di riprodurre le collisioni tra particelle, solitamente tramite un metodo Monte-Carlo.

L'algoritmo PIC pone l'ipotesi che le particelle cariche influenzino solo una porzione limitata di spazio, ovvero quello racchiuso dalla cella in cui si trova la particella in un dato istante temporale. Questa approssimazione ha un impatto importante a livello generale, poichè riduce la complessità algoritmica dell'interazione griglia-particelle a $O(n \log n)$, con n il numero di particelle, contrapposta alla complessità di n^2 caratteristica di problemi particellari più classici come quello degli N-corpi.

L'introduzione di una griglia spaziale porta ad uno smussamento dei campi e delle forze, poiché le loro eventuali fluttuazioni non sono osservabili su scale inferiori alla dimensioni di una cella della griglia. Per questo motivo, la grandezza caratteristica di una cella Δx deve essere abbastanza piccola da permettere risolvere con un numero sufficiente di nodi le lunghezze tipicamente considerate nel sistema, come:

- λ_D, lunghezza di Debye
- λ_{em}, lunghezza delle onde elettromagnetiche eventualmente interagenti col plasma
- λ_{sd}, profondità di penetrazione del plasma.

Il passo temporale di integrazione Δt è legato alla grandezza caratteristica di una cella Δx dalla condizione di stabilità di Courant–Friedrichs–Lewy. E' comunque possibile modificare il

ciclo sopra illustrato per permettere l'utilizzo di scale temporali differenti. Per esempio, gli elettroni (con frequenze relativamente elevate) possono essere integrati con un passo temporale Δ_{t_e} relativamente piccolo, mentre gli ioni con passo temporale Δ_{t_i} sensibilmente più lungo; a loro volta, anche i campi possono essere risolti su una terza scala Δ_{t_f}, breve per i campi elettromagnetici o più lunga per osservare effetti a bassa frequenza.

Proseguiamo con una disamina più accurata degli stadi di calcolo appena delineati.

3.2 Integrazione delle equazioni del moto: il metodo leapfrog

Le caratteristiche più importanti per un integratore da inserire in un software PIC sono velocità e bassa necessità di memoria, dovendo evidentemente operare su decine di migliaia (e spesso milioni) di particelle per migliaia di passi temporali. In quest'ottica, metodi di ordine elevato (come per esempio la famiglia di metodi Runge-Kutta), moltiplicando i passi intermedi all'interno di una singola iterazione in cambio di una precisione non sempre necessaria, risultano scarsamente attraenti.

Il metodo più diffuso prende il nome di *leapfrog*. Esso è un metodo del secondo ordine sufficientemente accurato per i nostri scopi, tuttavia richiede un numero di valutazioni di funzione per passo uguale a quello del metodo di Eulero, che però è solo del primo ordine.

I principali punti di forza dell'integrazione leapfrog sono la sua reversibilità temporale (è possibile integrare n passi in avanti e poi invertire la direzione di integrazione per tornare alle condizioni iniziali con altrettanti n passi) e la sua natura simplettica, ovvero la capacità di conservare l'energia dei sistemi dinamici (una proprietà che gli integratori Runge-Kutta non condividono). Si dimostra inoltre che l'errore del metodo scompare per un passo di integrazione $\Delta t \to 0$.

Esaminiamo brevemente il metodo: le due equazioni differenziali del primo ordine da integrare separatamente per ogni particella sono

$$\begin{aligned} m\frac{d\mathbf{v}}{dt} &= \mathbf{F} \\ \frac{d\mathbf{x}}{dt} &= \mathbf{v} \end{aligned} \tag{3.2}$$

dove $\mathbf{F}$ è la forza. Queste equazioni vengono sostituite con delle equazioni alle differenze finite

$$\begin{aligned} m\,\frac{\mathbf{v}_{new} - \mathbf{v}_{old}}{\Delta t} &= \mathbf{F}_{old} \\ \frac{\mathbf{x}_{new} - \mathbf{x}_{old}}{\Delta t} &= \mathbf{v}_{new} \end{aligned} \tag{3.3}$$

L'avanzamento dell'integratore è mostrato in Figura 3.1.

L'idea di base è avanzare $\mathbf{v}_t$ e $\mathbf{x}_t$ fino a $\mathbf{v}_{t+\Delta t}$ e $\mathbf{x}_{t+\Delta t}$, benchè $\mathbf{v}$ e $\mathbf{x}$ non siano conosciute allo stesso tempo. I valori di posizione e velocità vengono avanzati di un passo temporale, ma sono sfasati tra loro di mezzo passo, con le posizioni che precedono le velocità, e si scavalcano vicendevolmente all'avanzare dell'integrazione. In quest'ottica, sono necessari due accorgimenti: primo, all'istante iniziale $t = 0$ la velocità $\mathbf{v}(0)$ viene retrocessa a $\mathbf{v}(-\Delta t/2)$ usando la forza $\mathbf{F}(t = 0)$; secondo, le energie calcolate da $\mathbf{v}$ (cinetica) e $\mathbf{x}$ (potenziale o di campo) devono essere calibrate per apparire al medesimo istante.

Illustriamo ora un'altra particolarità del metodo leapfrog nel contesto di nostro interesse. Si consideri una particella carica in moto lungo l'asse x di uno spazio cartesiano (x,y,z). La particella sia immersa in un campo elettrico $\mathbf{E} = (\hat{x}E_x, 0, 0)$ ed in un campo magnetico statico ed

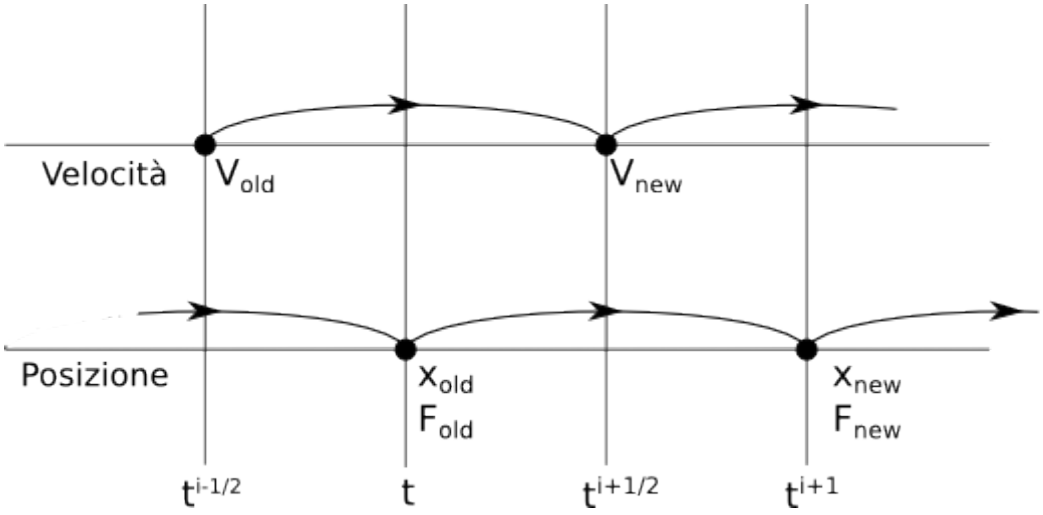


Figura 3.1: Schema del metodo di integrazione leapfrog

uniforme B_0 diretto esclusivamente lungo l'asse z. La forza agente sulla particella è chiaramente la forza di Lorentz

$$\mathbf{F} \quad = \quad q\mathbf{E} + q(\mathbf{v} \times \mathbf{B}). \tag{3.4}$$

La componente della forza dovuta al campo magnetico, in queste ipotesi, è semplicemente una *rotazione* di $\mathbf{v}$, ovvero il vettore delle velocità non cambia in modulo. La componente elettrostatica $q\mathbf{E}$ altera invece il modulo di v. L'operazione di integrazione delle velocità può quindi essere scomposta in tre parti. Indicando con t' e t'' due istanti temporali generici tali che $t - \Delta t/2 < t' < t'' < t - \Delta t/2$, e scegliendo come riferimento la scala temporale delle posizioni, possiamo distinguere:

1. *Mezza accelerazione*

$$\begin{aligned} v_x(t') &= v_x\left(t - \frac{\Delta t}{2}\right) + \left(\frac{q}{m}\right) E_x(t) \left(\frac{\Delta t}{2}\right) \\ v_y(t') &= v_y\left(t - \frac{\Delta t}{2}\right) \end{aligned} \tag{3.5}$$

2. *Rotazione*

$$\begin{pmatrix} v_x(t'') \\ v_y(t'') \end{pmatrix} = \begin{pmatrix} \cos\omega_c\Delta t & \sin\omega_c\Delta t \\ -\sin\omega_c\Delta t & \cos\omega_c\Delta t \end{pmatrix} \begin{pmatrix} v_x(t') \\ v_y(t') \end{pmatrix} \tag{3.6}$$

3. *Mezza accelerazione*

$$\begin{aligned} v_x\left(t + \frac{\Delta t}{2}\right) &= v_x(t'') + \left(\frac{q}{m}\right) E_x(t) \left(\frac{\Delta t}{2}\right) \\ v_y\left(t + \frac{\Delta t}{2}\right) &= v_y(t'') \end{aligned} \tag{3.7}$$

L'angolo di rotazione è

$$\Delta\theta = -\omega_c\Delta t, \tag{3.8}$$

dove ω_c è la frequenza di ciclotrone 2.16.

Come descritto in precedenza, all'istante iniziale, $\mathbf{v}(0)$ è retrocessa a $\mathbf{v}(-\Delta t/2)$, ruotando $\mathbf{v}(0)$ di un angolo $\Delta\theta = +\omega_c \Delta t/2$ e poi applicando una mezza accelerazione usando $-\Delta t/2$ ed il campo $\mathbf{E}(0)$ ottenuto da $\mathbf{x}(0)$.
A fronte di quanto detto finora, riscriviamo le equazioni 3.3 con una discretizzazione alle differenze finite centrali, usando come riferimento la scala temporale delle velocità e indicando con i l'iterazione corrente, in modo che risulti $t = i * \Delta t$:

$$\frac{\mathbf{x}^{i+\frac{1}{2}} - \mathbf{x}^{i-\frac{1}{2}}}{\Delta t} = \mathbf{v}^i, \tag{3.9}$$

$$\frac{\gamma^{i+1}\mathbf{v}^{i+1} - \gamma^i \mathbf{v}^i}{\Delta t} = \frac{q}{m}\left(\mathbf{E}^{i+\frac{1}{2}} + \bar{\mathbf{v}}^{i+\frac{1}{2}} \times \mathbf{B}^{i+\frac{1}{2}}\right) \tag{3.10}$$

dove $\gamma = 1/\sqrt{1 - v^2/c^2}$ è il fattore relativistico della particella, con c la velocità della luce. Per chiudere il sistema, $\bar{\mathbf{v}}^{i+\frac{1}{2}}$ deve essere espressa come funzione delle altre quantità. La soluzione di più largo impiego è quella data da Boris:

$$\bar{\mathbf{v}}^{i+1/2} = \frac{\gamma^i \mathbf{v}^i + \gamma^{i+1}\mathbf{v}^{i-1}}{2\bar{\gamma}^{i+1/2}}. \tag{3.11}$$

Il sistema può essere risolto in maniera molto efficiente seguendo il metodo di Boris, che prevede un disaccoppiamento degli spostamenti dovuti al campo elettrostatico e al campo magnetico, analogamente a quanto già descritto nelle (3.5)-(3.7). Si evita così di risolvere esplicitamente per il fattore relativistico $\bar{\gamma}^{i+1/2}$, che nello schema di Boris è dato da

$$\begin{aligned} \bar{\gamma}^{i+1/2} &= \sqrt{1 + \left(\gamma^i \mathbf{v}^i + \frac{q\Delta t}{2m}\mathbf{E}^{i+1/2}\right)} \\ &= \sqrt{1 + \left(\gamma^{i+1}\mathbf{v}^{i+1} - \frac{q\Delta t}{2m}\mathbf{E}^{i+1/2}\right)} \end{aligned} \tag{3.12}$$

3.2.1 Variante di Vay al leapfrog classico

È stato recentemente dimostrato da Vay che il classico metodo di Boris non preserva alcune proprietà fondamentali di conservazione delle forze e trasformazione di queste passando da un sistema di riferimento inerziale ad un altro.

In particolare, applicando le (3.10)-(3.12) al caso in cui i contributi reciproci del campo elettrostatico e del campo magnetico si annullino (cioè $\mathbf{E} + \mathbf{v} \times \mathbf{B} = 0$), si trova che una particella subisce una forza spuria nel caso generale in cui $\mathbf{E} \neq 0$ e $\mathbf{B} \neq 0$.

Prendiamo quindi in considerazione la soluzione proposta dallo stesso Vay e consideriamo invece di (3.11) la seguente media delle velocità:

$$\bar{\mathbf{v}}^{i+1/2} = \frac{\mathbf{v}^i + \mathbf{v}^{i+1}}{2} \tag{3.13}$$

così che la (3.10) diventa

$$\frac{\mathbf{v}^{i+1} - \mathbf{v}^i}{\Delta t} = \frac{q}{m}\left(\mathbf{E}^{i+1/2} + \frac{\mathbf{v}^i + \mathbf{v}^{i+1}}{2} \times \mathbf{B}^{i+\frac{1}{2}}\right). \tag{3.14}$$

Si dimostra che in questo modo l'update alle velocità è libero dalla presenza di forze spurie come invece osservato nell'integratore di Boris.

Per risolvere la (3.14) poniamo

$$\begin{aligned} \mathbf{u} &= \gamma \mathbf{v} \\ \mathbf{u}' &= \mathbf{u}^{\mathbf{i}} + \frac{q\Delta t}{m}\left(\mathbf{E}^{i+1/2} + \frac{\mathbf{v}^i}{2} \times \mathbf{B}^{i+1/2}\right), \end{aligned} \tag{3.15}$$

ottenendo

$$\mathbf{u}^{i+1} = \mathbf{u}' + \frac{q\Delta t}{m}\left(\frac{\mathbf{u}^{u+1}}{2\gamma^{i+1}} \times \mathbf{B}^{i+1/2}\right), \tag{3.16}$$

e definiamo anche le velocità a metà dei passi temporali, quando anche le posizioni sono note:

$$\begin{aligned} \mathbf{u}^{i+1/2} &= \mathbf{u}^i + \frac{q\Delta t}{2m}\left(\mathbf{E}^{i+1/2} + \mathbf{v}^i \times \mathbf{B}^{i+1/2}\right) \\ &= \mathbf{u}^{i+1} - \frac{q\Delta t}{2m}\left(\mathbf{E}^{i+1/2} + \mathbf{v}^{i+1} \times \mathbf{B}^{i+1/2}\right). \end{aligned} \tag{3.17}$$

Questo ci permette di separare l'algoritmo in due mezzi passi:

1. **Primo mezzo passo:** Ottenere $\mathbf{u}^{i+1/2}$ da $\mathbf{u}^i$ usando la prima delle (3.17);
2. **Secondo mezzo passo:** Ottenere $\mathbf{u}^{i+1}$ da $\mathbf{u}^{i+1}$ con la seguente sequenza di operazioni:

$$\begin{aligned} \mathbf{u}' &= \mathbf{u}^{i+1/2} + \left(\frac{q\Delta t}{2m}\mathbf{E}^{i+1/2}\right) \\ \tau &= \left(\frac{q\Delta t}{2m}\mathbf{B}^{i+1/2}\right) \\ u^* &= \mathbf{u}' \cdot \tau / c \\ \gamma' &= \sqrt{1 + \frac{u'^2}{c^2}} \\ \sigma &= \gamma'^2 - \tau^2 \\ \gamma^{i+1} &= \sqrt{\frac{\sigma + \sqrt{\sigma^2 + 4(\tau^2 + u^{*2})}}{2}} \\ \mathbf{t} &= \tau / \gamma^{i+1} \\ s &= \frac{1}{(1 + t^2)} \\ \mathbf{u}^{i+1} &= s[\mathbf{u}' + (\mathbf{u}' \cdot \mathbf{t})\mathbf{t} + \mathbf{u}' \times \mathbf{t}]. \end{aligned} \tag{3.18}$$

Si dimostra che l'update delle velocità risultante dall'integratore di Vay non presenta la forzo spuria osservata invece adottando il metodo di Boris.

3.3 Interpolazione e deposizione: funzioni di forma

La rappresentazione della reciproca influenza tra griglia e particelle è affidata alle due operazioni chiamate *deposizione* ed *interpolazione*. La prima ha lo scopo di calcolare la densità di carica sui nodi della griglia dalle posizioni delle particelle, mentre la seconda di calcolare i valori dei campi elettromagnetici sulle particelle quando il valore dei campi stessi è noto solo sui nodi della griglia.

Entrambe queste operazioni presuppongono l'attribuzione alla particella di un certo numero di *pesi*, ovvero di coefficienti che indichino quanto una particella influenza determinati nodi della griglia e viceversa. Determinare i pesi implica una qualche forma di interpolazione tra i nodi della cella in cui si trova una determinata particella (ricordiamo che uno dei presupposti,

nonchè punti di forza, dell'algoritmo PIC è che le particelle influenzino solo lo spazio della cella che le contiene). Si dimostra che è necessario usare gli stessi pesi sia per la deposizione che per l'interpolazione per ragioni di conservazione di carica e per evitare un'auto-forza (una particella accelera sé stessa).

I pesi, equivalenti a delle medie spaziali, sono definiti usando delle funzioni di forma g per le particelle.

Nel caso dell'interpolazione, la forza media su una particella n è come già detto la forza di Lorentz (3.4), che si può riscrivere in funzione delle fasi servendosi della seconda delle (2.3):

$$\mathbf{F}(\mathbf{x}, \mathbf{p}_n, t) = q\left(\mathbf{E}(\mathbf{x}) + \frac{\mathbf{p}_n \times \mathbf{B}(\mathbf{x})}{m\gamma c}\right); \tag{3.19}$$

allora, la forza media interpolata sulla particelle sarà

$$\overline{\mathbf{F}}_n(\mathbf{x}, \mathbf{p}_n, t) = \int g(\mathbf{x} - \mathbf{x}_n)\mathbf{F}(\mathbf{x}, \mathbf{p}_n, t)d\mathbf{x}. \tag{3.20}$$

Indicizzando i nodi della cella corrente C con la funzione caratteristica $\chi_\imath$, di l'indice multidimensionale $\imath = (i, j, k)$, è possibile decomporre l'integrale in una somma finita di medie:

$$\overline{\mathbf{F}}_n(\mathbf{x}, \mathbf{p}_n, t) = \sum_{\imath \in C} \int_{\chi_\imath} g(\mathbf{x} - \mathbf{x}_n)\mathbf{F}(\mathbf{x}, \mathbf{p}_n, t)d\mathbf{x}. \tag{3.21}$$

La forza è data dai campi, che sono discretizzati in modo da avere un valore singolo e costante per ogni nodo. È pertanto possibile scrivere gli integrali di cella come funzione della sola posizione della particella.:

$$\overline{\mathbf{F}}_n(\mathbf{x}, \mathbf{p}_n, t) = \sum_{\imath \in C} F_\imath(\mathbf{p}_n, t) \int_{\chi_\imath} g(\mathbf{x} - \mathbf{x}_n)d\mathbf{x}, \tag{3.22}$$

o, introducendo i *fattori di forma* $S_\imath(\mathbf{x}_n) = \int_{\chi_\imath} g(\mathbf{x} - \mathbf{x}_n)d\mathbf{x}$,

$$\overline{\mathbf{F}}_n(\mathbf{x}, \mathbf{p}_n, t) = \sum_{\imath \in C} F_\imath(\mathbf{p}_n, t)S_\imath(\mathbf{x}_n). \tag{3.23}$$

Per i fattori di forma, che di fatto sono i pesi da attribuire alle particelle, vale la proprietà

$$\sum_\imath S_\imath(\mathbf{x}) = 1. \tag{3.24}$$

Questa proprietà risulta giustificata e intuitivamente ovvia nel contesto della conservazione delle quantità fisiche.

Nel caso della deposizione, invece, essendo la densità di carica $\rho(\mathbf{x})$ definita come $\rho(\mathbf{x}) = \sum_n qg(\mathbf{x} - \mathbf{x}_n)$, dove n sono tutte le particelle contenute negli elementi della griglia in cui partecipa il nodo $\imath$, si ha:

$$\begin{aligned} \rho_\imath &= \frac{\int_{\chi_\imath} \rho(\mathbf{x})d\mathbf{x}}{\int d\mathbf{x}_{\chi_\imath} = V_\imath} = \int_{\chi_\imath} \left[\sum_n qg(\mathbf{x} - \mathbf{x}_n)\right] d\mathbf{x}/V_\imath \\ &= \sum_n q\left[\int_{\chi_\imath} g(\mathbf{x} - \mathbf{x}_n)\right]/V_\imath = \frac{1}{V_\imath}\sum_n qS_\imath(\mathbf{x}_n) \end{aligned} \tag{3.25}$$

Descriviamo qualitativamente i criteri più comuni con i quali si determinano le funzioni ed i fattori di forma (ed in definitiva i pesi) per le particelle:

- Nella *pesatura di ordine zero* si conta semplicemente il numero di particelle entro una distanza $\Delta x/2$ dal nodo j-esimo (ricordiamo che Δx è la grandezza caratteristica di una cella) e si assegna quel numero (sia, per esempio, $N(j)$) al nodo; allora, in una dimensione la densità di carica in quel punto sarà $n_j = N(j)/\Delta x$. Un nome comune per questa tecnica di pesatura è *nearest-grid-point* o *NGP* e computazionalmente è molto veloce, poiché richiede un'unica lettura dei dati della griglia.

  Adottando questo schema, quando una particella entra nella prossimità di un nodo, la densità di carica aumenterà a gradino di una quantità corrispondente alla carica della particella; quando la particella si allontana dal nodo, la densità di carica diminuirà istantaneamente della medesima quantità finita. Osserviamo due effetti. Il primo è che la particella appare avere una *forma rettangolare* di larghezza Δx. Questo porta noi (e la griglia) a credere di avere un'insieme di *particelle dalle dimensioni finite*; dunque, si osserverà un comportamento fisico di tali particelle, invece di particelle puntiformi, anche se ciò difficilmente altera gli effetti di base del plasma oggetto di studio. Il secondo effetto è che le variazioni a gradino delle grandezze fisiche al passaggio delle particelle attraverso una cella produrranno una distribuzione di densità ed un campo elettromagnetico con una quantità di rumore tale da essere inaccettabile in molte applicazioni legate al plasma. È dunque necessario cercare un pesatura migliore.

- La *pesatura del primo ordine* smussa le fluttuazioni di densità e campi, riducendo il rumore ma richiedendo maggiori risorse di calcolo, accedendo (in una dimensione) a due nodi per ogni particella, due volte per passo temporale. È possibile vedere questa scelta come un miglioramento nell'uso di particelle di dimensione finita o come una migliore interpolazione. Le particelle cariche ora appaiono come *nuvole* di dimensione finita che possono attraversarsi liberamente. Se consideriamo la nuvola nominale di densità uniforme e di diametro Δx, monodimensionalmente la parte della carica totale q_c assegnata al nodo j, di coordinata X_j, sarà

$$q_j = q_c \left[\frac{\Delta x - (x_i - X_j)}{\Delta x} \right]. \tag{3.26}$$

  L'effetto è quello di produrre un fattore di forma triangolare di ampiezza $2\Delta x$. Si fa notare che l'assegnazione di una carica puntiforme in x_i ai nodi più vicini attraverso un'interpolazione lineare porterebbe allo stesso risultato (ed è proprio da questo fatto che deriva il nome *particle-in cell*). Muovendosi attraverso la griglia, la nuvola contribuisce alla densità molto più regolarmente rispetto al metodo NGP, risultando in densità di plasma e campi molto meno rumorosi, che permettono una griglia più grossolana, ed un numero minore di particelle nella simulazione. Questo schema di pesi è accettabile per la maggior parte dei problemi di simulazione del plasma.

  Si fa notare che i pesi calcolati al primo ordine (o, in maniera equivalente, con una interpolazione lineare della posizione della particella rispetto ai nodi della cella in cui si trova) coincidono con le coordinate baricentriche della particella rispetto alla sua cella di appartenenza. Questo aspetto torna particolarmente utile nell'implementazione di un algoritmo PIC in un codice di calcolo (per ulteriori approfondimenti, si vedano le Sezioni 6.2 e 6.6).

- Una *pesatura di ordine più elevato*, tramite l'uso di polinomi quadratici, cubici e spline arrotonda ulteriormente le irregolarità del fattore di forma delle particelle, riduce il rumore su densità e campi, e tende a diminuire gli effetti non fisici, al costo di un maggior numero di operazioni richieste.

Le Figure 3.2 e 3.3 illustrano graficamente funzioni e fattori di forma monodimensionali, centrati e normalizzati dei primi 4 ordini.

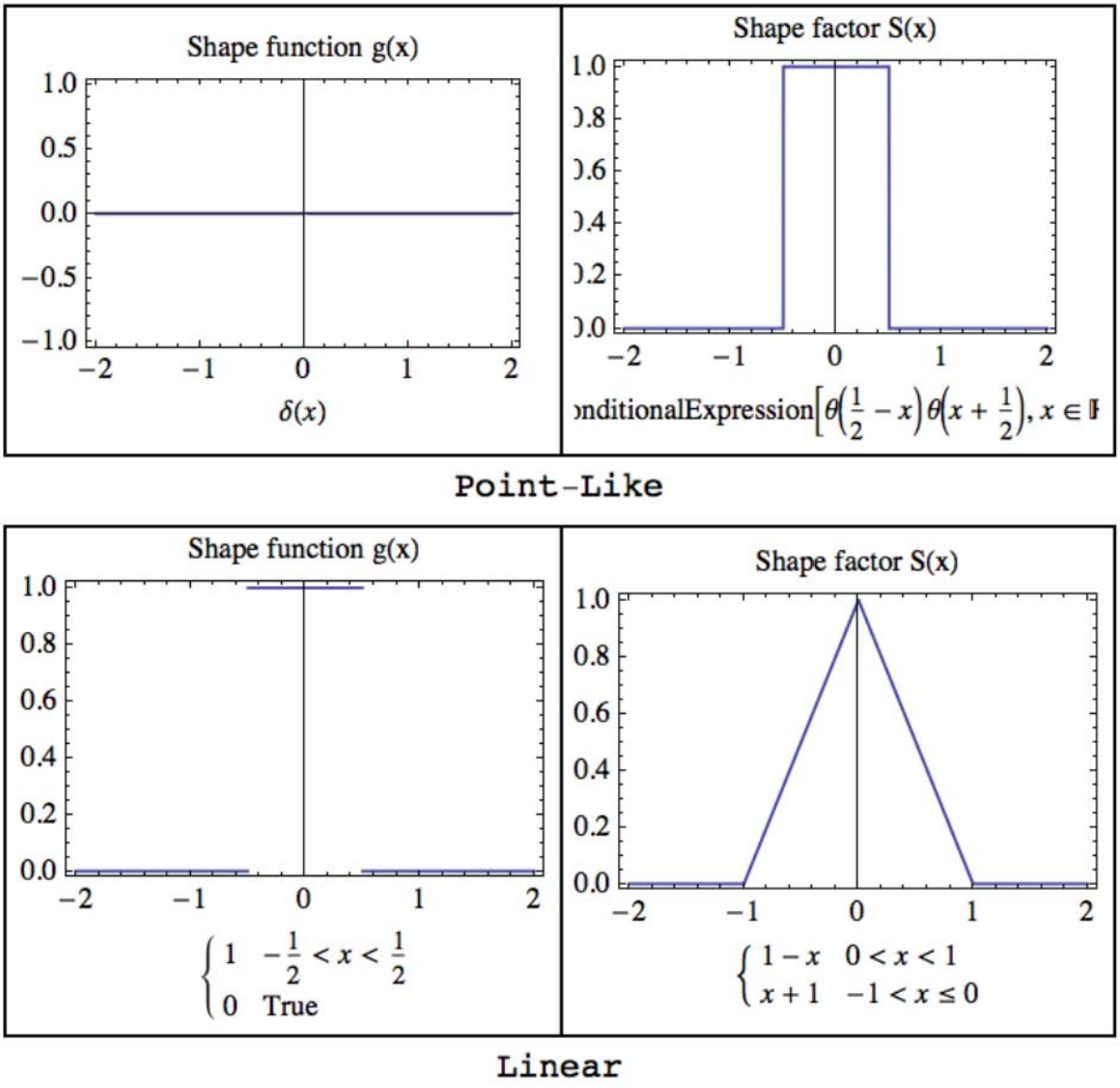
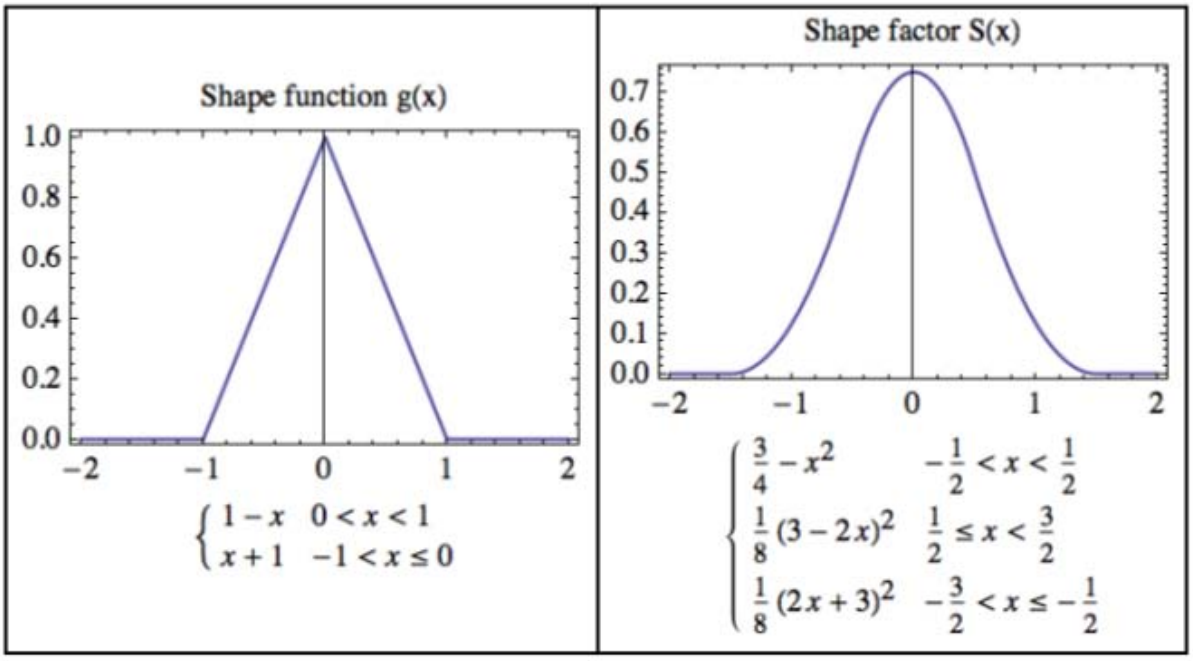


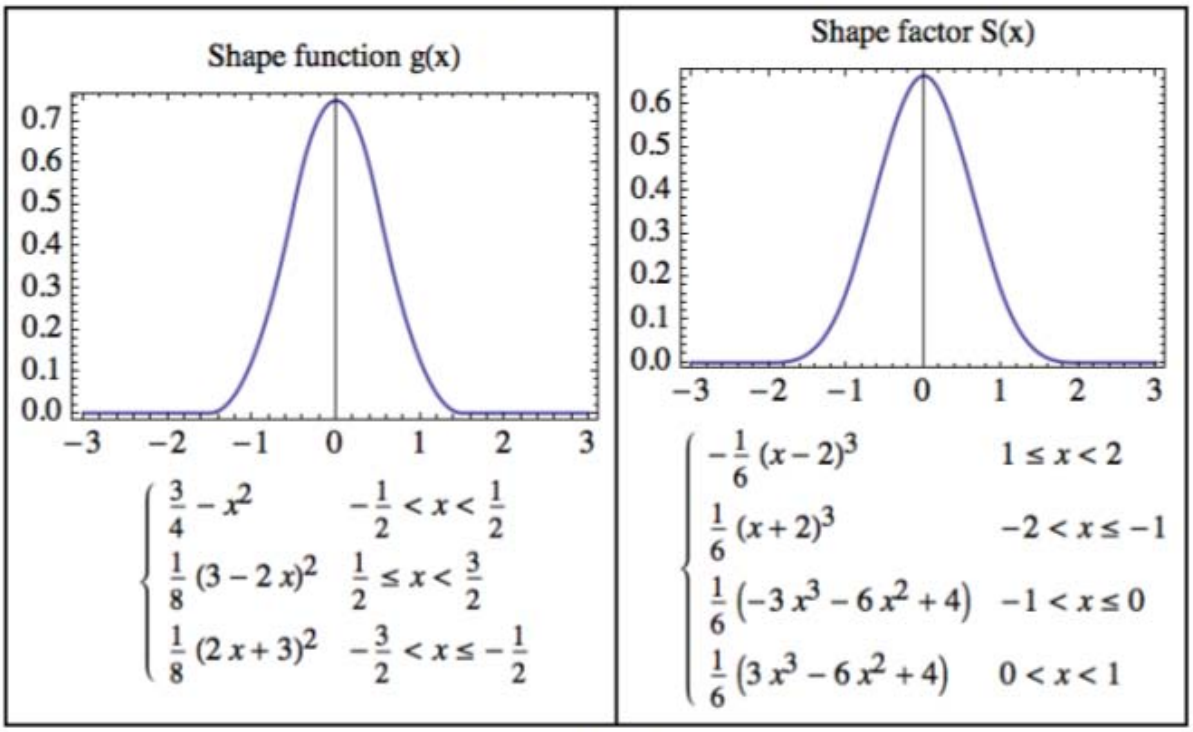
Figura 3.2: Funzioni di forma e fattori di forma di ordine zero e di primo ordine

3.4 Integrazione delle equazioni di campo

Una volta determinate le densità di carica e di corrente su tutti i nodi della griglia, è necessario calcolare i campi elettrici e magnetici, in generale risolvendo le equazioni di Maxwell (2.3). Questo compito può essere assolto servendosi degli usuali strumenti matematici e numerici per la soluzione di sistemi differenziali, come metodi alle differenze finite (di facile implementazione nel caso di un dominio di calcolo strutturato) o i più potenti e flessibili (ma considerevolmente più complessi) metodi agli elementi finiti (FEM), metodi spettrali e così via.



Quadratic



Cubic

Figura 3.3: Funzioni di forma e fattori di forma di secondo e terzo ordine

Nel caso di un problema elettrostatico, la procedura si semplifica risolvendo invece l'equazione di Poisson (2.6), che riportiamo di seguito per completezza:

$$\nabla^2 \Phi = -\frac{\rho}{\varepsilon_0}. \tag{3.27}$$

Sul nodo arbitrario j di una griglia monodimensionale possiamo scrivere le seguenti espressioni alle differenze finite:

$$E_j = \frac{\Phi_{j-1} - \Phi_{j+1}}{2\Delta x} \tag{3.28}$$

$$\frac{\Phi_{j-1} - 2\Phi_j + \Phi_{j+1}}{(\Delta x)^2} = -\frac{\rho_j}{\varepsilon_0} \tag{3.29}$$

Quest'ultima può essere scritta in forma matriciale come

$$\mathbf{A}\Phi = -\frac{(\Delta x)^2}{\varepsilon_0}\rho. \tag{3.30}$$

Usiamo i ρ_j noti alle coordinate x_j per ottenere l'incognita Φ_j e poi E_j, per j che va da 0 a $L/\Delta x$, dove L è la lunghezza del sistema di N nodi. Applicando le note condizioni al contorno a $x = 0, L$ e tutti i ρ_j, otteniamo un sistema con un numero di equazioni pari a quello di incognite, e dunque risolvibile.

Un approccio molto potente, nel caso di sistemi periodici, consiste nell'usare una trasformata di Fourier discreta per tutte le quantià della griglia. Questa strategia fornisce anche informazioni spaziali sullo spettro di ρ, ϕ ed E, che sono utili per correlare i risultati alla teoria del plasma, e permettono inoltre di controllare lo spettro delle quantità di campo.

3.5 Il pacchetto di simulazione F3MPIC

F3MPIC è un codice di simulazione del plasma fondato sull'algoritmo Particle-in-Cell, con una griglia tridimensionale di elementi tetraedrici accoppiata con un solutore agli Elementi Finiti (nel dominio del tempo) ed un solutore elettromagnetico (nel dominio della frequenza). Grazie alla natura non strutturata della griglia di calcolo, può gestire geometrie di complessità arbitraria, con un numero arbitrario di specie fisiche, sia cariche che neutre. È possibile importare la geometria da simulare attraverso un generico software CAD tridimensionale. F3MPIC è stato sviluppato per l'ottimizzazione e la progettazione di dettaglio di propulsori helicon per veicoli spaziali. Il codice comprende funzionalità avanzate come elettroni di Boltzmann, e la simulazione delle collisioni con neutri tramite metodi Monte-Carlo (MCC), ed è stato estensivamente validato su molteplici piattaforme.

Capitolo 4

Introduzione a CUDA

Spinta dall'insaziabile richiesta del mercato di grafica 3D ad alta definizione ed in tempo reale, l'Unità di Elaborazione Grafica (Graphic Processing Unit o GPU) che normalmente si trova nei Personal Computer si è evoluta in un processore altamente parallelo, dotato di tremenda potenza di calcolo ed una elevatissima larghezza di banda di memoria, come mostrato nelle Figure 4.1 e 4.2, che confrontano l'evoluzione in prestazioni delle GPU NVIDIA e delle CPU (Central Processing Unit) Intel negli ultimi anni.

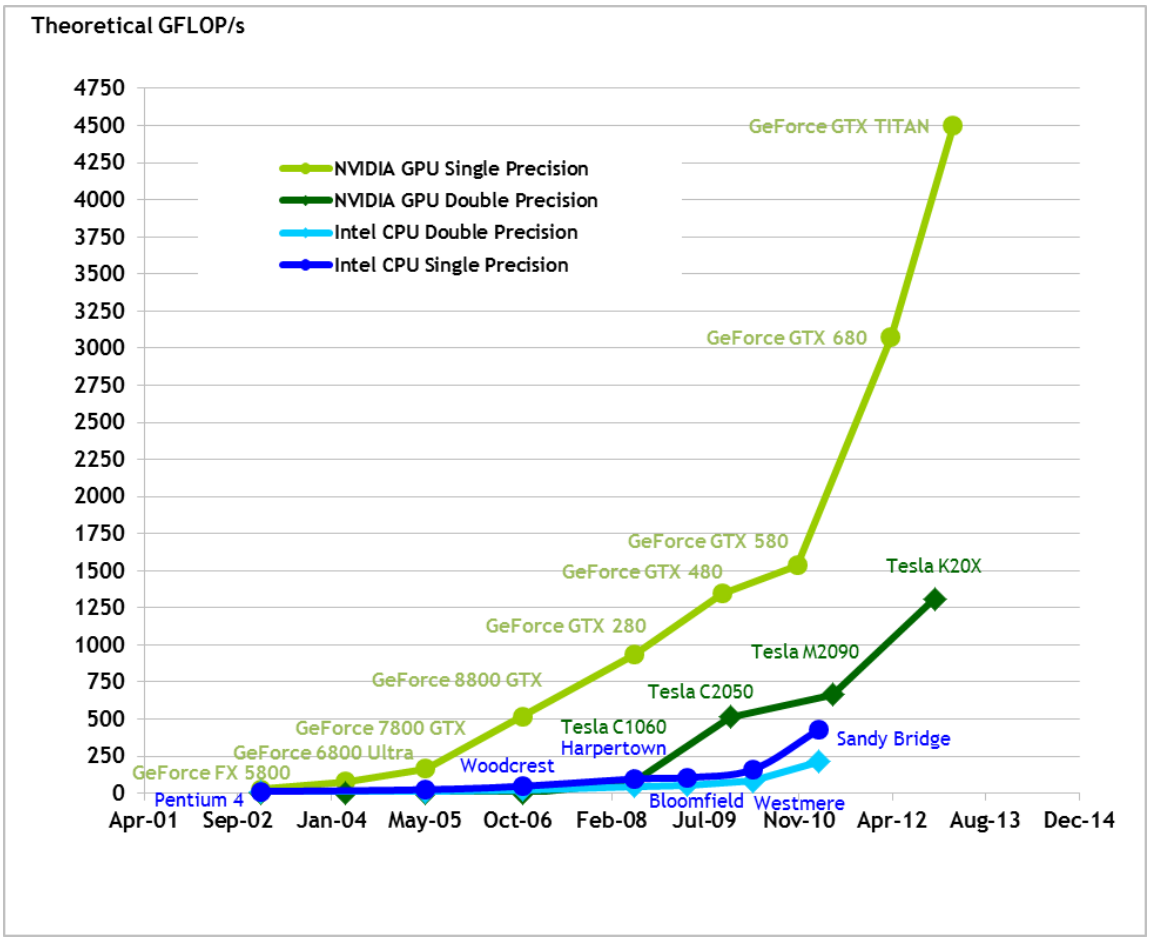


Figura 4.1: Prestazioni di calcolo in virgola mobile per la GPU e la CPU

La ragione dietro questa differenza di capacità di calcolo in virgola mobile tra CPU e GPU

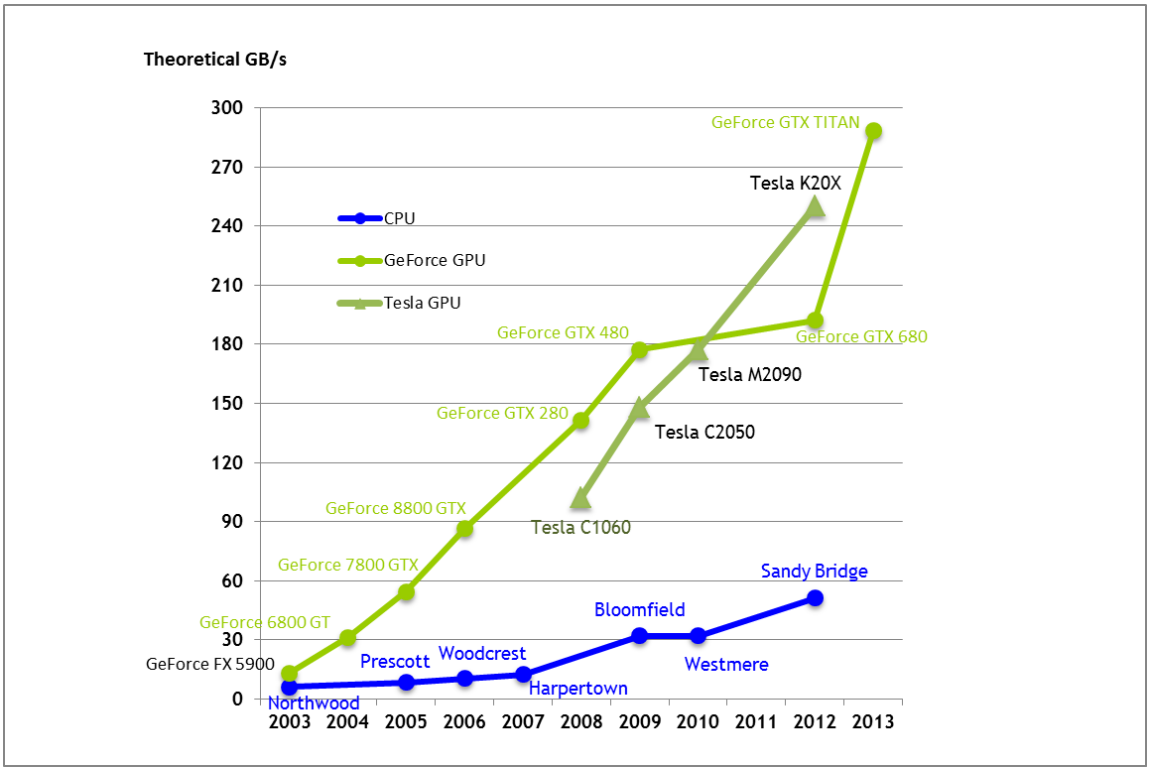


Figura 4.2: Banda di memoria per la GPU e CPU

è che la GPU è specializzata per calcoli altamente paralleli e dall'alta densità computazionale – esattamente ciò di cui si tratta il rendering grafico – e quindi progettata in modo che più transistor siano dedicati all'elaborazione di dati piuttosto che al data caching ed al controllo di flusso.

In maniera più specifica, la GPU si adatta particolarmente bene ad affrontare problemi che possono essere espressi come computazioni data-parallel (lo stesso programma viene eseguito su molti dati diversi in parallelo) ad alta intensità aritmetica (il rapporto fra operazioni aritmetiche ed operazioni di memoria). Siccome il programma viene eseguito per ogni elemento di dati, ci sono minori requisiti di un sofisticato controllo di flusso, e siccome è eseguito su molti elementi di dati ed ha alta intensità aritmetica, la latenza di accesso alla memoria può essere nascosta con calcoli invece di grandi cache di dati.

Nel Novembre 2006, NVIDIA (uno dei più grandi produttori mondiali di GPU) ha introdotto CUDA^{TM}(acronimo per Compute Unified Device Architecture), un modello di calcolo parallelo a scopo generico che sfrutta le GPU NVIDIA per risolvere molti problemi computazionalmente complessi in maniera più efficiente che con una tradizionale CPU. Più semplicemente, CUDA permette di eseguire sulla GPU programmi a scopo generico (cioè non strettamente legati alla visualizzazione di un output grafico), sfruttando le enormi potenze di calcolo racchiuse nell'unità grafica (che viene esposta come un processore parallelo), senza passare attraverso la programmazione di API grafiche. Infatti, CUDA introduce delle astrazioni che permettono di esprimere il parallelismo in maniera semplice, ma comunque completo, gerarchico e scalabile; il primo e principale strumento per scrivere programmi CUDA sono delle apposite estensioni al linguaggio C, ma sono disponibili anche interfacce a linguaggi come FORTRAN o OpenCL.

Il calcolo eterogeneo, ovvero la possibilità di coadiuvare una tradizionale CPU seriale con coprocessori paralleli specializzati (siano essi GPU o altre architetture many-core), assegnando a ciascun processore i carichi di lavoro ad esso più adatti, negli ultimi anni ha fatto passi avanti

decisivi, che lo stanno portando rapidamente alla ribalta, soprattutto nel campo delle scienze applicate. Nello svolgimento del presente lavoro, CUDA è stato preferito ad altre soluzioni per le seguenti motivazioni.

- E' accessibile: programmi realizzati per un'architettura CUDA possono essere eseguiti su tutte le GPU realizzate da NVIDIA dal 2007 ad oggi, in una vasta gamma di prodotti che va dagli acceleratori grafici per l'intrattenimento e l'uso professionale, fino a processori per il calcolo ad alte prestazioni (HPC). Un altro fattore determinante in quest'ottica è che a parità di prestazioni hardware, le soluzioni basate su dispositivi CUDA sono notevolmente meno costose di soluzioni più tradizionali basate su CPU.

- E' relativamente semplice: non solo è possibile scrivere software usando un'API che estende il popolare linguaggio C, ma le astrazioni di CUDA fanno sì che il problema da parallelizzare possa essere tradotto in un programma in modo molto più immediato e lineare rispetto ad una programmazione multithread su CPU, e che questo programma scali il suo parallelismo in modo trasparente all'aumentare del numero dei core di elaborazione.

- CUDA è disponibile da più di 6 anni, ed in questo periodo ha raggiunto la maturità da molti punti di vista: quello hardware (per funzionalità offerte dai dispositivi), quello software (stabilità dei driver, disponibilità di software di terze parti come librerie e framework) ed infine quello degli strumenti di sviluppo (ambienti di sviluppo integrati pensati specificamente per lavorare con le GPU sono gratuitamente disponibili per tutti i sistemi operativi di maggior impiego.).

- CUDA ha una naturale predisposizione per il calcolo scientifico: l'elaborazione data-parallel è usata sì nel rendering di grafica 3D, nell'elaborazione delle immagini e nella codifica/decodifica video, ma anche moltissimi problemi scientifici ed ingegneristici possono essere espressi in maniera quasi immediata come problemi data-parallel: fra questi, l'elaborazione dei segnali, le simulazioni fisiche, e la biologia computazionale.

- Risulta più conveniente di entrambe le soluzioni concorrenti esistenti ad oggi:

  - OpenCL: Si tratta di un modello aperto per scrivere programmi destinati ad architetture eterogenee di varia natura, e quindi non ristretto alle GPU. È stato introdotto nel 2008 da Apple ed è mantenuto e sviluppato dal consorzio Khronos Group, già conosciuto per la manutenzione dell'interfaccia di programmazione grafica OpenGL. Nonostante adotti un'impostazione e delle astrazioni molto simili a quelle di CUDA, OpenCL presenta diversi svantaggi. Prima di tutto, le sue molteplici piattaforme di destinazione e la gestione da parte di un organismo molto eterogeneo ne rende gli aggiornamenti molto poco tempestivi, con ovvie conseguenze negative sulle funzionalità a disposizione. Inoltre, OpenCL richiede che il programmatore gestisca direttamente molte caratteristiche dell'esecuzione di un programma che in CUDA invece vengono curate automaticamente dal componente runtime; per questo motivo, alcuni autori definiscono addirittura OpenCL come un linguaggio di livello più basso rispetto alle interfacce CUDA. Infine, numerosi esempi in letteratura dimostrano che un programma OpenCL è spesso più lento del suo analogo in CUDA.
  - Intel MIC: Incalzata dalla rapida adozione di coprocessori GPU nel campo del calcolo ad altre prestazioni (HPC), Intel ha ufficialmente presentato nel novembre 2012 la sua architettura per coprocessori destinati a piattaforme eterogenee. Denominata Intel MIC (abbreviazione per *Intel Many Integrated Core Architecture*), consiste in un elevato numero di core di elaborazione x86 (circa una cinquantina per i prodotti

attualmente in commercio), collegati tra loro da un bus bidirezionale ultra-veloce ad anello e una cache L2 coerente. Uno dei punti di forza determinanti secondo Intel, è la possibilità di rimuovere la curva di apprendimento necessaria per utilizzare efficacemente architetture GPU, poichè, essendo MIC basata su x86, la parallelizzazione del codice sarebbe gestita quasi interamente in modo automatico dal processo di compilazione e dal componente runtime. Nonostante le promettenti premesse in termini di funzionalità e prestazioni, ed il loro sicuro ruolo da protagonisti nel prossimo futuro dell'HPC, i prodotti basati su Intel MIC sono ancora di recentissima introduzione, devono dimostrare la loro competitività in molti campi di applicazione, mettere a disposizione soddisfacenti risorse software per gli sviluppatori e raggiungere una sufficiente disponibilità sul mercato.

4.1 Modello software

L'interfaccia di programmazione CUDA C estende il normale linguaggio C permettendo al programmatore di definire funzioni C, chiamate *kernel* che, quando chiamate, vengono eseguite N volte in parallelo da N differenti CUDA thread, invece che una volta sola come normali funzioni C.

Un kernel è definito usando la specifica di dichiarazione `__global__` ed il numero di CUDA thread che eseguono un kernel per una determinata chiamata è specificato usando la nuova sintassi di configurazione d'esecuzione (*execution configuration*) `<<<...>>>` . Ad ogni thread che esegue un kernel è assegnato un identificatore di thread (thread ID) che è accessibile dall'interno del kernel attraveros la variabile predefinita `threadIdx`.

Attraverso la execution configuration, il programmatore può arbitrariamente raggruppare i thread in blocchi di thread (*thread blocks*) monodimensionali, bidimensionali o tridimensionali. Questo fornisce un modo naturale di invocare calcoli attraverso elementi in un dominio come un vettore, una matrice o un volume. C'è un limite al numero di thread per blocco, dal momento che tutti risiedono sullo stesso processore di elaborazione e devono condividerne le limitate risorse di memoria. Su GPU di ultima generazione, un blocco di thread può contenere fino a 1024 thread. Tuttavia, un kernel può essere eseguito da molteplici blocchi di uguali dimensioni, così che il numero totale di thread è uguale al numero di thread per blocco moltiplicato per il numero di blocchi.

I blocchi possono a loro volta essere organizzati in una griglia monodimensionale, bidimensionale o tridimensionale di blocchi, come illustrato in Figura 4.3. Il numero di blocchi in una griglia è solitamente dettato dalle dimensioni dei dati che devono essere elaborati, e viene anch'esso specificato attraverso la execution configuration. All'interno della griglia, i blocchi e le loro dimensioni vengono identificati dalle variabili predefinite `blockIdx` e `blockDim`.

È richiesto che i blocchi siano eseguiti indipendentemente. Deve essere possibile eseguirli in qualsiasi ordine, in parallelo o in serie. Il requisito di indipendenza consente ai blocchi di thread di essere pianificati in qualsiasi ordine attraverso qualsiasi numero di processori, permettendo ai programmatori di scrivere codice che scali in modo automatico e trasparente con il numero di processori. I thread all'interno di un blocco possono cooperare condividendo dati attraverso una memoria condivisa (*shared memory*) e sincronizzando la loro esecuzione per coordinare gli accessi alla memoria; i punti di sincronizzazione all'interno di un kernel vengono specificati con la funzione intrinseca `__syncthreads()`.

Durante la loro esecuzione, i CUDA thread possono accedere a dati che risiedono in molteplici spazi di memoria (come mostrato in Figura 4.4):

- Registri: sono locazioni di memoria individuali per ogni thread ed offrono la più alta

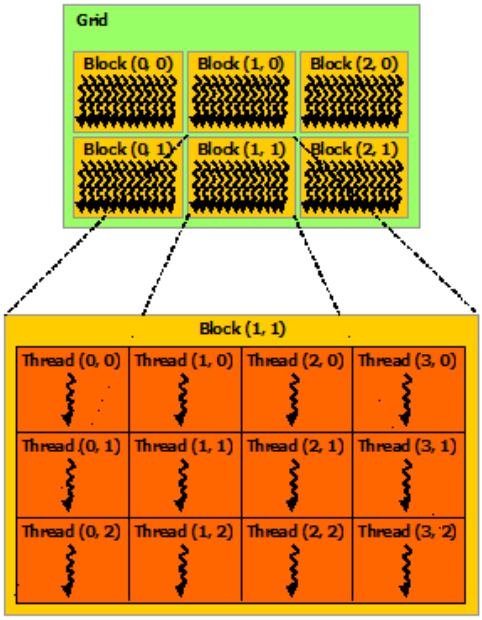


Figura 4.3: Organizzazione dei thread in blocchi e griglia di calcolo

performance possibile, ma sono limitati ed un uso eccessivo può causare lo spostamento dei dati in una memoria più lenta (fenomeno del *register spilling*).

- Shared memory: È una memoria presente sul chip della GPU, veloce e visibile a tutti i thread di un blocco, ma da gestire con attenzione perchè relativamente limitata
- Global memory: Risiede sulla DRAM del dispositivo ed è il tipo di memoria più generico ed esteso, ma anche il più lento. Tuttavia, se i dati rispettano specifici requisiti di allineamento o di accesso, le letture di un gruppo di thread vengono condensate in un'unica istruzione (*coalesced access*). Soddisfare questi requisiti non è sempre banale e perciò è spesso difficile ottenere una performance ottimale dalla memoria globale.
- Constant memory: Si tratta di una memoria ristretta, dotata di cache ed a sola lettura, ma con una particolarità. Se tutti i thread leggono contemporaneamente la stessa posizione di constant memory, il dato ivi contenuto viene trasmesso in un'unica transazione a gruppi di 16 thread (si dice pertanto che la memoria costante ha capacità di *broadcasting*).
- Texture memory: È un tipo di memoria dotata di cache ed ottimizzata per accessi che rispettino una daterminata località dei dati (bidimensionale o tridimensionale). Deve essere gestita con funzioni specifiche e per le sue particolarità trova scarso impiego in applicazioni che non riguardino la visualizzazione grafica e l'elaborazione di immagini e video.

Le memorie globale, costante e texture sono ottimizzate per usi differenti e sono persistenti tra i vari lanci di kernel di una stessa applicazione.

Come già anticipato, CUDA rappresenta un modello di calcolo eterogeneo (*Heterogeneous Computing*): esso assume che i CUDA thread siano eseguiti su un dispositivo (*Device*) fisicamente separato che opera come un coprocessore per il sistema ospite (*Host*) che esegue il programma C. Questo è il caso, per esempio, di kernel che vengono eseguiti su una GPU ed il resto del programma C è eseguito dalla CPU. Il modello di programmazione CUDA assume anche che entrambi Host e Device mantengano separati i propri spazi di memoria DRAM

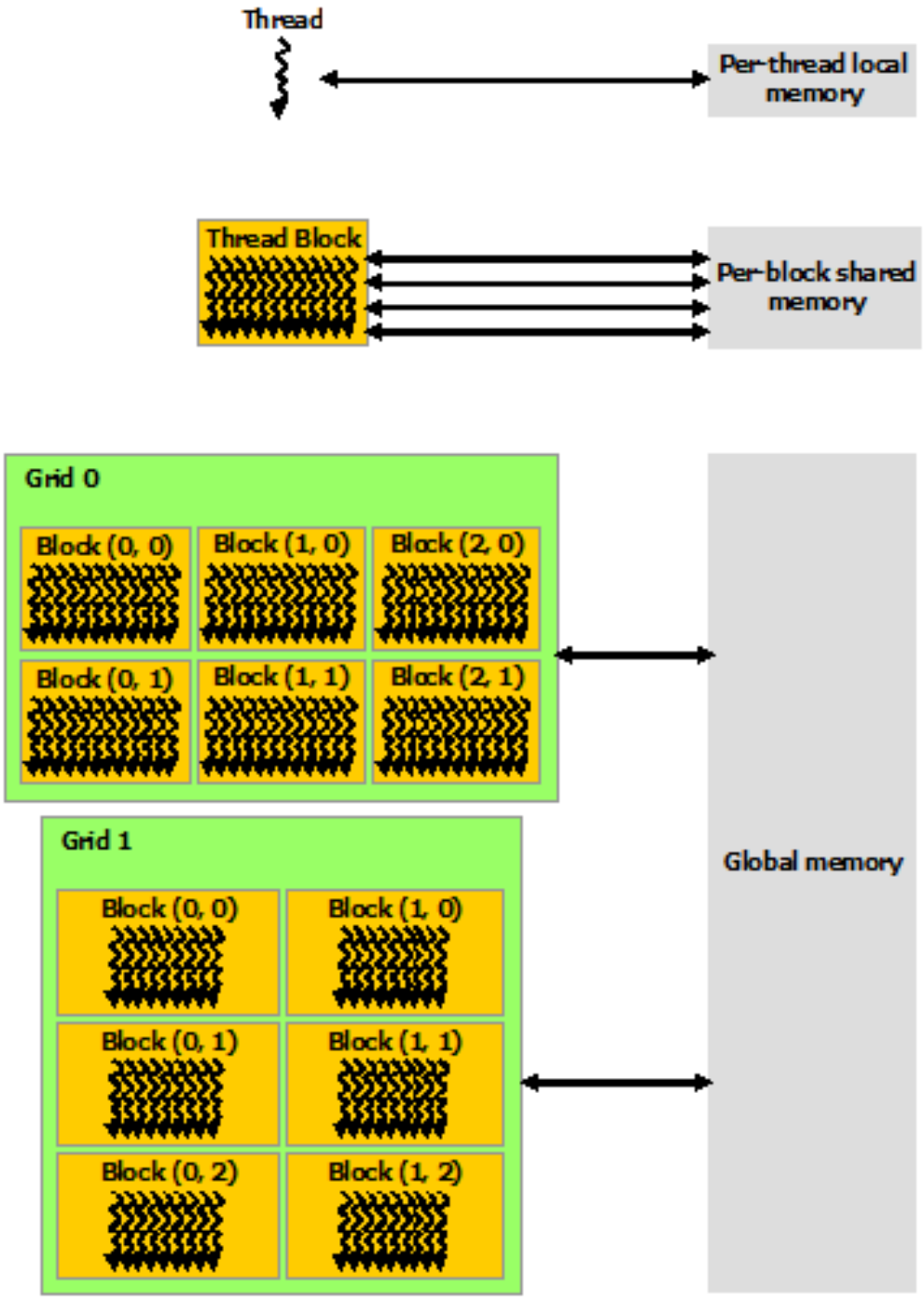


Figura 4.4: Gerarchia di memoria CUDA

(Dynamic Random Access Memory), chiamati rispettivamente Host memory e Device memory. Dunque, un programma gestisce gli spazi di memoria globale, costante e texture (che risiedono sul dispositivo) attraverso invocazioni al componente a tempo di esecuzione di CUDA, richiedendo per esempio allocazioni, deallocazioni e trasferimenti di dati tra memoria Host e memoria Device.

Il calcolo eterogeneo permette asincronia e collaborazione tra i vari componenti del sistema: a ciascun processore può essere assegnato il carico di lavoro più adeguato alle sue caratteristiche.

4.2 Modello hardware

L'architettura CUDA è costruita attorno ad una struttura scalabile di *Streaming Multiprocessors (SM)* capaci di multithreading. Quando un programma CUDA nella CPU dell'Host invoca la griglia di un kernel, i blocchi della griglia sono numerati e distribuiti ai multiprocessori disponibili per l'esecuzione. I thread di un blocco sono eseguiti simultaneamente su un multiprocessore, e diversi blocchi possono essere eseguiti simultaneamente su un multiprocessore. Quando alcuni blocchi terminano la loro esecuzione, nuovi blocchi vengono lanciati sui multiprocessori vacanti.

Un multiprocessore è progettato per eseguire centinaia di thread contemporaneamente. Per gestire un così elevato numero di thread, esso impiega un'architettura unica chiamata SIMT (*Single-Instruction, Multiple-Thread*): benchè concettualmente simile alla più famosa SIMD (Single-Instruction, Multiple-Data), ne è distinta dal fatto che una singola istruzione non viene eseguita su più dati, ma viene inviata a più thread, i quali provvedono ad eseguirla individualmente.

Il multiprocessore crea, gestisce, pianifica ed esegue i thread in gruppi di 32 thread paralleli chiamati *warps*. I singoli thread che compongono un warp partono dallo stesso indirizzo di programma, ma hanno la possibilità di diramarsi ed eseguirsi indipendentemente. Quando ad un multiprocessore sono assegnati uno o più blocchi di thread da eseguire, esso li partiziona in warps, la cui esecuzione viene pianificata da un *warp scheduler*. Un warp esegue un'istruzione comune alla volta, perciò la piena efficienza si realizza quando tutti i 32 thread del warp sono concordi nel loro percorso di esecuzione. Se dei thread in un warp divergono a causa di una condizione dipendente dai dati, il warp esegue serialmente ogni percorso di esecuzione, disabilitando i thread che non ne fanno parte, e quando tutti i percorsi terminano, i thread convergono nuovamente verso un'esecuzione comune. Questa divergenza si verifica solo all'interno di un warp, dal momento che warp differenti vengono eseguiti in modo indipendente anche se seguono sequenza di istruzioni differenti.

Il contesto di esecuzione (contatori di programma, registri, ecc.) per ogni warp elaborato da un multiprocessore è mantenuto sul chip durante l'intero periodo di vita del warp. In particolare, ogni multiprocessore ha un set di registri a 32 bit, che sono partizionati tra i warp, ed una memoria condivisa (shared memory) che come già illustrato è partizionata tra i blocchi di thread.

Il numero di blocchi e di warp che possono risiedere contemporaneamente su un multiprocessore per un dato kernel dipende dall'ammontare di registri e shared memory usati dal kernel e dall'ammontare di registri e shared memory disponibili sul multiprocessore. Sono anche imposti un numero massimo di blocchi e di warp residenti su un multiprocessore. Tutti questi limiti sono dipendenti dall'architettura del dispositivo,che viene riassunta, assieme alle funzionalità di calcolo, in una sigla identificativa chiamata *Compute Capability*.

Se non ci sono abbastanza registri o shared memory disponibili in un multiprocessore per elaborare almeno un blocco, il lancio del kernel fallirà.

Si descrive brevemente di seguito l'architettura NVIDIA CUDA di ultima generazione, so-

prannominata “Kepler” e, nella sua forma più completa, identificata dalla Compute Capability 3.5 . In questa architettura, un multiprocessore è composto da:

- 192 CUDA Cores per operazioni aritmetiche con interi e numeri in virgola mobile;
- 32 Special Function Units per funzioni trascendentali in singola precisione;
- 4 Warp Schedulers;
- 64KB di cache, configurabile come 48KB di shared memory e 16KB di cache L1, o viceversa;
- 64000 registri a 32 bit.

I multiprocessori sono raggruppati in *Graphics Processing Clusters (GPC)*, ciascuno dei quali comprende un numero variabile di multiprocessori, a seconda dell'architettura della GPU. Le GPU di Compute Capability 3.5 sono dotate anche di 1536KB di cache L2 condivisa da tutti i thread, e possono ospitare fino a 1024 thread per blocco, oppure 16 blocchi per multiprocessore, e possono eseguire 64 warp contemporaneamente per multiprocessore, per un totale massimo di 2048 thread residenti per SM. Le GPU che implementano l'architettura Kepler nel pieno delle sue capacità sono quelle designate come GK110. In questa configurazione, la GPU presenta fino a 14 Streaming Multiprocessor, 5 Graphics Processing Cluster e 2688 CUDA Cores.

4.3 Caratteristiche avanzate

Introduciamo in questa Sezione alcuni aspetti avanzati di CUDA che hanno rivestito un ruolo importante nel nostro lavoro e che meritano un breve approfondimento per una migliore comprensione.

4.3.1 CUDA Dynamic Parallelism

Il Dynamic Parallelism è un'estensione al modello di programmazione CUDA che permette ad un kernel di lanciare e sincronizzare a sua volta nuovi kernel (e associate griglie di calcolo) direttamente sulla GPU. La creazione dinamica di parallelismo in qualunque punto un programma la richieda offre entusiasmanti nuove opportunità.

L'abilità di generare lavoro direttamente sulla GPU può ridurre la necessità di trasferire il controllo dell'esecuzione tra host e device, poichè decisioni riguardanti le configurazioni di lancio possono essere fatte a tempo di esecuzione dai thread che vengono eseguiti sul coprocessore. In aggiunta, la creazione di lavoro parallelo dipendente dai dati direttamente all'interno di un kernel può avvalersi degli scheduler e dei bilanciatori di carico presenti a livello hardware nella GPU, in maniera dinamica e adattiva.

Tutto ciò permette un'espressione più semplice di algoritmi e schemi di programmazione che prevedano ricorsioni, strutture di cicli irregolari, e altri costrutti difficili da trasporre in un singolo livello di parallelismo. Diventa anche possibile eseguire funzioni delle librerie associate a CUDA in maniera simile ad una modalità batch, per esempio lanciando operazioni di algebra lineare su un arbitrario numero di matrici di piccole/medie dimensioni.

Da un punto di vista di programmazione, l'estensione al modello CUDA che abilita il Dynamic Parallelism consiste nell' introduzione di un nuovo componente, chiamato *device runtime*, che è un sottoinsieme funzionale dell'host runtime tradizionale. Il device runtime permette di servirsi, dall'interno dei kernel, delle stesse funzioni e della stessa sintassi normalmente usate

per il lancio di kernel, i trasferimenti di memoria intra-dispositivo, la sincronizzazione dell'esecuzione e la gestione del dispositivo, con minime eccezioni che tengono conto del differente ambiente di esecuzione.

Il Dynamic Parallelism è supportato esclusivamente da dispositivi di Compute Capability 3.5 e superiori.

4.3.2 CUDA Streams

Gli *stream* sono uno strumento che permette alle applicazioni di gestire esecuzioni simultanee e concorrenti. Uno stream è una sequenza di comandi che viene eseguita in ordine, ma stream differenti possono eseguire i loro comandi indipendentemente l'uno dall'altro.

Distribuendo accuratamente le istruzioni a differenti streams è possibile scambiare dati con il sistema host mentre la GPU sta eseguendo un kernel, oppure permettere l'esecuzione simultanea di più kernel (se il dispositivo lo consente).

4.3.3 Thrust

Thrust è una libreria di template C++ per CUDA, basata sul modello della Standard Template Library (STL). Thrust consente di implementare applicazioni parallele dalle performance elevate con uno sforzo di programmazione ridotto, tramite un'interfaccia di alto livello, comunque pienamente compatibile ed interoperabile con CUDA C.

Thrust mette a disposizione una ricca collezione di primitive data-parallel come scan, sort e reduce, che possono essere impiegate per implementare algoritmi complessi in un codice sorgente conciso e leggibile. Descrivendo i compiti da eseguire in termini di astrazioni di alto livello si lascia a Thrust la libertà di selezionare automaticamente l'implementazione più efficiente. Di conseguenza, Thrust può essere utilizzato nella prototipazione rapida di applicazioni CUDA, dove conta la produttività, ma anche in codice di produzione, dove la robustezza e la performance assoluta sono cruciali.

4.4 Linee guida di programmazione

Le linee guida per ottenere delle buone performance con un programma CUDA si possono riassumere in 3 punti:

- Massimizzare il parallelismo per raggiungere la massima occupazione dei multiprocessori;
- Ottimizzare l'uso di memoria per massimizzarne il flusso;
- Ottimizzare l'uso di istruzioni per massimizzarne il flusso.

Il parallelismo si può ricercare a molti livelli (di applicazione, di dispositivo...), ma il più imperativo da realizzare è sicuramente quello all'interno di un multiprocessore: è a questo livello che bisogna padroneggiare due concetti fondamentali per il calcolo su GPU, ovvero *occupancy* e *latency hiding*. Ogni volta che deve impartire un'istruzione, il warp scheduler di un multiprocessore seleziona un warp pronto ad eseguire la sua prossima istruzione ed impartisce quest'ultima ai thread attivi del warp. Il tempo che un warp impiega per essere pronto ad eseguire la sua prossima istruzione è chiamato *latenza*, e per alcune istruzioni può essere ingente (accessi alla memoria globale hanno una latenza che va dai 400 agli 800 cicli di clock). Un pieno utilizzo si realizza quando tutti i warp scheduler hanno sempre la possibilità di impartire istruzioni a warp disponibili durante i periodi di latenza di altri warp, in altre parole quando la latenza è del tutto nascosta dall'abbondanza di unità esecutive (da qui il termine *latency hiding*). Ne

consegue che un parametro importante dell'efficacia operativa della GPU è dato dal numero di warp residenti contemporaneamente in un multiprocessore; il rapporto tra questo numero ed il massimo di warp presenti in uno stesso multiprocessore viene chiamato *occupancy* e per mantenerlo il più elevato possibile è necessario gestire oculatamente le limitate risorse hardware della GPU, come i registri e la shared memory. È altrettanto importante diminuire le latenze, accedendo preferibilmente a memorie on-chip e usando ove possibile istruzioni intrinseche, che hanno meno precisione ma richiedono un numero minore di cicli per essere eseguite.

Ottimizzare il flusso di memoria ha principalmente due significati: il primo è ridurre il più possibile i trasferimenti a bassa larghezza di banda, non solo quelli tra Host e Device, tenendo il più possibile i dati sul device, ma anche quelli tra memoria globale e la GPU vera e propria, poiché la DRAM di una scheda grafica ha sì una banda superiore rispetto alla memoria di un computer tradizionale, ma rimane comunque molto lenta rispetto alle velocità di accesso di shared memory e, soprattutto, dei registri. Il secondo è usare gli spazi di memoria appropriati per le loro caratteristiche: dati utili a tutti i thread vanno trasferiti in shared memory, dati che non devono essere modificati e tutti i thread accederanno agli stessi indirizzi sono appropriati per la memoria costante, e infine dati che presentano una corrispondenza spaziale al loro posizionamento è opportuno riporli nella memoria texture.

Ottimizzare l'uso di istruzioni riguarda l'uso tecniche anche abbastanza raffinate, ma ne esistono comunque alcune importantissime da applicare a livello base. Ogni istruzione di controllo di flusso (`if`, `else`, `while`, `for`, `switch`, `do`) può influenzare significativamente il flusso di istruzioni inducendo dei thread di uno stesso warp a divergere (cioè a seguire differenti percorsi di esecuzione). Se questo succede, i differenti percorsi di esecuzione devono essere serializzati, aumentando il numero totale di istruzioni eseguite per il warp e degradando la performance. Quando tutti i differenti percorsi di esecuzione sono completi, i thread convergono nuovamente verso un percorso comune. Per quanto appena detto, evitare la divergenza dei warp è uno dei primi obiettivi da tenere presente quando si progetta un kernel CUDA. Infine, va posta particolare attenzione al posizionamento delle barriere di sincronizzazione, realizzate con `__syncthreads()`. Infatti, un errato posizionamento può rallentare la performance di un kernel o, molto peggio, se posta in un'istruzione condizionale che non viene valutata in modo identico da tutti i thread di un warp, può bloccare perennemente l'esecuzione di un kernel, facendo aspettare in modo indefinito alcuni thread per altri che ormai sono già "andati avanti".

Parte II

Implementazione di moduli PIC su GPU

Capitolo 5

Lo stato dell'arte

L'impostazione di base di un metodo Particle-in-Cell per la simulazione del plasma prevede che le particelle non interagiscano mai direttamente tra di loro, ma attraverso le modifiche indotte nei campi elettromagnetici dal loro movimento: si prevede dunque una certa indipendenza nell'evoluzione dello stato di ogni singola particella. Questo ed altri aspetti portano ad intuire che l'algoritmo PIC possa essere ben espresso in un programma CUDA. La possibilità di sfruttare le ampie capacità di calcolo delle GPU è peraltro molto desiderabile, in quanto la simulazione del plasma esige considerevoli risorse e anche riprodurre un sistema di medio-piccole dimensioni per un tempo rilevante a fini scientifici e pratici può richiedere da alcune ore a pochi giorni di tempo-macchina.

Certamente, non siamo i primi ad aver fatto questa osservazione. Proprio per questo motivo, prima di cominciare a progettare un nuovo software adatto alle nostre esigenze, è stato condotto un approfondito e scrupoloso studio sulla letteratura relativa alla simulazione di plasma sulla GPU, per ricercare lo stato dell'arte ed individuare pratiche comuni, strategie e soluzioni da sfruttare, o da cui trarre ispirazione. Riassumiamo in questo Capitolo i lavori che hanno segnato delle tappe importanti in questo campo, o che hanno avuto una rilevante influenza sul nostro lavoro di tesi.

La prima pubblicazione degna di nota, e che ha acquisito un ruolo seminale nella materia, è dovuta a Stantchev et al. [5], nella quale si identifica subito lo stadio di aggiornamento della densità di carica come il più delicato da parallelizzare, sviluppando comunque una soluzione molto efficiente. Questa prevede la definizione di gruppi di celle della griglia(*cell clusters*) e il riordinamento (a cui nel calcolo parallelo si fa riferimento con il termine *sorting*) delle particelle dipendentemente dal cluster di appartenenza. I cell clusters sono abbastanza piccoli da essere contenuti nella memoria condivisa e l'update viene accumulato semi-atomicamente con una tecnica ormai obsoleta di *thread tagging* (al tempo della pubblicazione non erano ancora disponibili operazioni atomiche in CUDA). La semplicità e l'efficacia di questo metodo ha portato alla sua adozione in molte altre implementazioni PIC su GPU. Stantchev ha anche identificato il problema di realizzare un adeguato sorting delle particelle per il loro metodo di accumulazione delle densità, e propone una soluzione molto interessante, per certi versi simile ad un algoritmo Quicksort con un grande uso della shared memory sulla GPU. I cell clusters sono creati seguendo un partizionamento gerarchico del dominio di calcolo, rappresentato da un albero BSP (Binary Space Partitioning), e ordinati secondo una curva a riempimento di spazio, come una curva di ordine Z (*Z-order curve*, anche detta Z-ordering o, in alcuni casi, Morton code). Questo permette al sorting di percorrere ricorsivamente i vari rami dell'albero: una volta arrivati in fondo, tutte le particelle si troveranno nel loro cluster di appartenenza. Stantchev ha dimostrato questo metodo in due dimensioni, ma fa subito notare come sia immediatamente estendibile alle 3 dimensioni. Il miglioramento di prestazioni viene quantificato come compreso

tra 10 e 20 volte, a seconda dei casi, rispetto ad un analogo software su CPU.

Altri importanti contributi sono venuti da Kong et al [6]. Evidenziando i limiti della tecnica di thread tagging di Stantchev, propongono, qualora le operazioni atomiche native non siano disponibili, una soluzione alternativa basata sull'uso di memoria dichiarata `volatile` (un qualificatore disponibile in CUDA). Questa tecnica è anche in grado di sostituire efficacemente operazioni atomiche in doppia precisione, non ancora direttamente disponibili sulle GPU NVIDIA. Kong nota inoltre che, per le proprietà matematiche e fisiche della simulazione, le particelle non percorrono mai più di una determinata distanza ad ogni passo temporale, e molte rimangono nello stesso cell cluster. Di conseguenza, un sorting completo non è necessario. Viene proposto un meccanismo di riordinamento solo fra le celle che attraversano i bordi dei cluster: queste vengono scritte in porzioni "cuscinetto" dei buffer per le particelle, che vengono adattate a tempo di esecuzione a seconda delle esigenze. L'algoritmo è fortemente dipendente da dalla frazione di particelle che ad ogni iterazione attraversano le pareti dei cluster, e lavora al meglio se le particelle si muovono al più di un cluster alla volta, pena la necessità di altri passaggi di sorting. Inoltre la scelta delle porzioni cuscinetto nella memoria riservata alle particelle è opinabile: se troppo ristretta inizialmente, le operazioni di ampliamento avverranno frequentemente; se troppo estese, si spreca prezioso spazio di memoria. Kong dichiara comunque miglioramenti dalle 27 alle 81 volte, dipendenti dalla temperatura del plasma, rispetto all'analogo CPU

Un'esperienza estremamente interessante è quella descritta da Joseph et al [7], autori della riimplementazione del codice XGC1 su GPU. A differenza degli altri codici, XGC1 usa una griglia triangolare asimmetrica e non pone alcun limite al movimento delle particelle. Joseph riconosce subito che una siffatta mesh è molto più complicata da partizionare efficacemente, rispetto alle griglie strutturate usate dagli altri autori, e procede dunque ad un accurato confronto teorico e pratico di strategie per il partizionamento delle entità di simulazione e dei carichi di lavoro: alcuni schemi favoriscono la contiguità dei dati, altri l'equilibrio della mole di lavoro tra le varie unità di calcolo. Ogni partizione della griglia comprende un contorno di "celle guardia", che vengono replicate tra differenti partizioni e facilitano la fase di deposizione, anche per la quale vengono valutate differenti strategie. I contributi più importanti risiedono comunque nelle considerazioni sulla gestione dei carichi di lavoro per la GPU e nella applicabilità generale delle metodologie sviluppate.

La tesi di Payne [8] offre un'ampia ed approfondita panoramica dell'implementazione di codici PIC su GPU, riportando anche pubblicazioni che nel presente documento si è scelto di trascurare. Il pregio maggiore del lavoro di Payne è tuttavia la valutazione accurata, quantitativa e critica delle più diffuse tecniche per ogni stadio di simulazione, alla luce di recenti strumenti hardware e software; Payne avanza anche la proposta della strategia di *stream compaction* per velocizzare collisioni e reiniezioni di particelle. Vengono dichiarati miglioramenti di prestazioni superiori alle 50 volte rispetto al software originale che si proponeva di parallelizare su GPU.

Per completezza, segnaliamo anche il lavoro di Rossi [9], con la sua implementazione del framework *jasmine* per la simulazione di plasma laser-accelerato, ed il lavoro presso l'Università di Dresda su PIConGPU [10], uno dei primissimi simulatori PIC su GPU ad essere implementato e scalato su molteplici nodi di un cluster.

Capitolo 6

Sintesi ed implementazione di un nuovo algoritmo

Lo scopo principale del nostro lavoro è realizzare dei componenti software che permettano al pacchetto F3MPIC di beneficiare della potenza di calcolo delle GPU, mantenendo intatte le sue capacità originali.

Da un punto di vista generale, tutti gli stadi dell'algoritmo PIC hanno le caratteristiche per poter trarre vantaggio dalla presenza del coprocessore grafico. Si è scelto tuttavia di non occuparsi al momento della soluzione delle equazioni di campo: a differenza del lavoro di vari autori, F3MPIC si avvale di un solutore agli elementi finiti (nello specifico, il software GetDP) per ricavare i campi elettromagnetici nei vari nodi della griglia. L'implementazione di un componente così delicato e complesso come un framework di soluzione di sistemi differenziali tramite il metodo FEM sarebbe andata chiaramente oltre gli scopi ed i limiti temporali della presente tesi. Si è altresì deciso di non portare sulla GPU le operazioni di generazione di nuove particelle, in quanto oltre ad essere complesse da parallelizzare, F3MPIC vi opera in concomitanza delle operazioni di estensione delle allocazioni di memoria, se necessarie, e queste al momento sono decisamente problematiche da tradurre sulla GPU.

Sempre generalmente parlando, è immediatamente chiaro, in accordo con quanto esposto nel Capitolo 5, che la parallelizzazione di un codice PIC in CUDA dovrà prevedere un'organizzazione spazialmente ordinata delle strutture di dati che costituiscono gli elementi della simulazione (griglia e particelle), per due ragioni: la prima, per assicurare un accesso coalesced alla memoria globale, nella quale i dati verranno ovviamente conservati per consistenza tra i lanci dei kernel; la seconda, per ridurre al minimo possibile la divergenza dei warp in esecuzione.

A queste considerazioni, le esigenze di F3MPIC aggiungono delle complicazioni notevoli. Infatti, una delle caratteristiche peculiari di F3MPIC è la presenza di una griglia non strutturata, tridimensionale e tetraedrica, pensata per adattarsi a geometrie arbitrariamente complesse. Essendo non strutturata, la griglia non può essere suddivisa in cluster di celle dalle medesime dimensioni e dai confini definiti e prevedibili. Inoltre, a differenza che con una mesh strutturata, non è banale sapere in quale cella si trovi una data particella in un qualsiasi istante temporale: è necessario introdurre uno stadio di calcolo dedicato al tracciamento delle particelle. Perdipiù, le relazioni tra gli indirizzi in memoria dei dati relativi alle particelle e quelli relativi alla griglia non possono preservare una località spaziale, compromettendo fortemente la velocità di accesso alla memoria globale. Aggirare questo ostacolo non è immediato, poichè si dimostra matematicamente che non è possibile stabilire un ordine di percorrenza univoco e senza ripetizioni di un insieme spaziale non strutturato. Nessun autore, tra quelli da noi consultati, si è confrontato con simili problematiche: quasi tutti i codici realizzati sono basati su mesh strutturate, e questo rende molte metodologie precedenti inutili ai nostri scopi. L'unico autore a lavorare con una

mesh non strutturata è stato Joseph, ma nel suo caso la griglia era bidimensionale. Possiamo comunque trarre utili suggerimenti dai nostri predecessori.

Per prima cosa dobbiamo decidere che prospettiva adottare nella scrittura del codice, se quella dei nodi o delle particelle. Assegnare un thread ad ogni particella sarebbe in accordo con la loro natura lagrangiana, tuttavia, per quanto appena detto, porterebbe ad una considerevole disuniformità negli accessi alla memoria e ad una riduzione della larghezza di banda. Per contro, assegnare un thread ad ogni nodo della griglia sembrerebbe rimediare a questo problema, poichè tutte le particelle visibili ad un nodo potrebbero essere disposte contiguamente. In realtà questo approccio ha molti aspetti negativi: primo, per le caratteristiche della mesh non strutturata, sarebbe impossibile trovare un ordinamento delle particelle in modo che tutti i nodi le vedano come blocchi continui di memoria; secondo, aggirare quanto appena detto implicherebbe creare spazi di memoria addizionali per registrare quante e quali particelle vengono viste da ciascun nodo; sarebbe impossibile determinare a priori lo spazio da dedicare a questi buffer, e sarebbe necessario aggiornarli ad ogni iterazione; terzo, in molti casi di nostro interesse, i nodi della mesh sono poche decine di migliaia, o meno: un simile numero è pari ai thread contemporaneamente attivi in una GPU GK110, correndo potenzialmente il rischio di non sfruttare a pieno il dispositivo e lasciarne inattiva una parte; invece, le particelle solitamente si contano nell'ordine dei milioni. Scegliamo quindi di far seguire ai thread l'evoluzione di singole particelle. Vedremo in seguito come si è cercato di attenuare la sparsità degli accessi alla memoria.

Per velocizzare l'interpolazione e la deposizione di carica, inizialmente si è presa in considerazione una soluzione simile a quella adottata da Li et al [11] per la soluzione di equazioni integrali di campi elettromagnetici. Li sovrappone una mesh strutturata non uniforme, rappresentata da un *octree*, alla originale mesh non strutturata, e usa i nodi della mesh non uniforme come passaggi intermedi per una doppia interpolazione dalla mesh non strutturata a dove sia necessario conoscere i valori dei campi. Questo approccio tuttavia funziona esclusivamente quando si lavora con grandezze di campo, che per loro natura sono liberamente interpolabili nello spazio; è risultato ben presto chiaro che una simile doppia interpolazione con una griglia strutturata ausiliaria avrebbe portato ad una violazione della conservazione di carica nella fase di deposizione.

Prendendo spunto dal lavoro di Joseph et al [7], anche noi riteniamo che la migliore soluzione per trattare con una mesh non strutturata e facilitare l'uso degli spazi di veloce memoria condivisa sia operare un partizionamento spaziale del dominio di calcolo, che tuttavia non può essere rigoroso e definito come con una mesh strutturata.

La costruzione e lo sfruttamento di questo partizionamento costituisce la caratteristica più innovativa (non tanto nella sostanza, quanto nel metodo) e più peculiare del nostro algoritmo, poichè ne costituirà la spina dorsale e si ripercuoterà in tutte le sue fasi.

Scegliamo di combinare i concetti di Li e di Joseph, sovrapponendo un octree alla griglia non strutturata, ma invece di usarlo per definire una griglia ausiliaria, lo impieghiamo come riferimento per partizionare il dominio.

Un octree è una struttura di dati appartenente alla categoria degli alberi, che costituisce un caso particolare, regolare ed esteso a 3 dimensioni di un albero BSP. Nel caso più comune ed usato, ogni nodo dell'albero equivale ad un sottinsieme cubico dello spazio, ma nulla proibisce che i nodi descrivano spazi a forma di parallelepipedo. Il nome octree deriva dal fatto che ogni nodo interno ha esattamente 8 figli, se ne è provvisto, ognuno dei quali racchiude uno spazio pari ad un ottavo del volume del genitore (il quale viene effettivamente diviso in 8 ottanti); i nodi senza figli vengono chiamati *leaf nodes* (letteralmente "nodi foglia"). Gli octree, come il loro equivalente bidimensionale *quadtree*, sono strutture che esprimono un partizionamento spaziale gerarchico, sono noti per la loro capacità di essere costruiti adattivamente ed ampiamente usati nel campo della computer grafica.

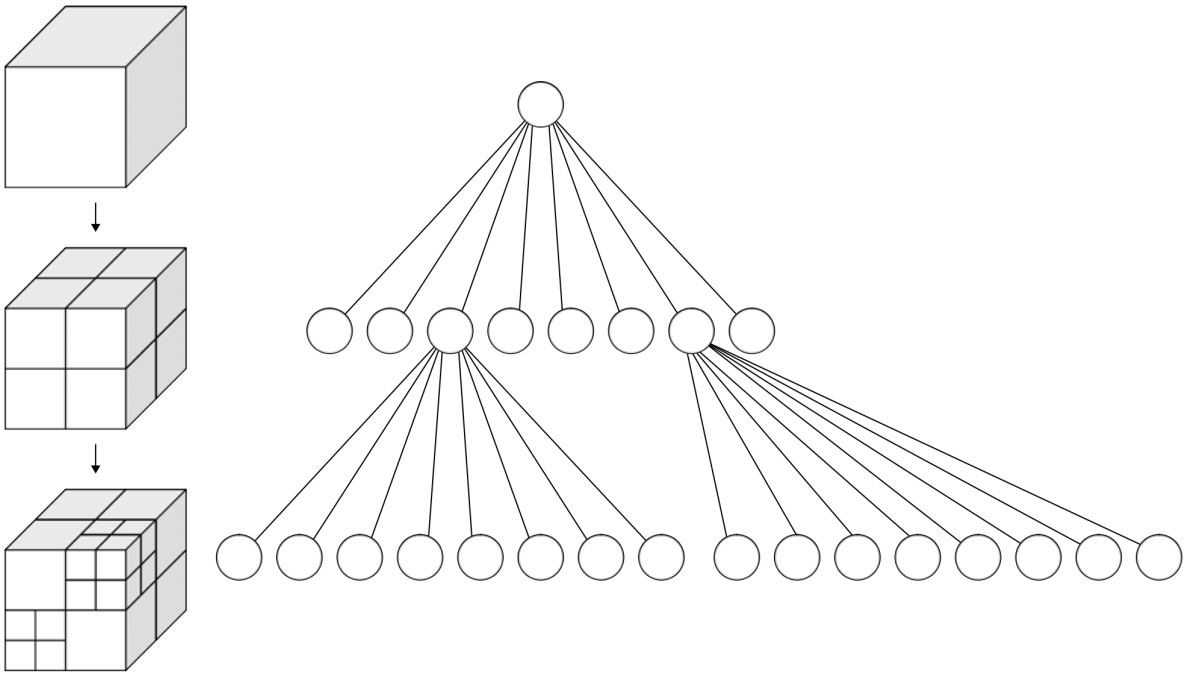


Figura 6.1: Rappresentazione grafica di un octree

La scelta di un octree è superiore ai vari partizionamenti proposti da Joseph, principalmente grazie al controllo ed alla flessibilità che si può esercitare nella sua costruzione, tuttavia i benefici che porta al nostro algoritmo sono molti più di quanti ne appaiano a prima vista. Infatti un octree:

- È costruito adattivamente sulla base della mesh originaria, conformandosi al meglio a qualsiasi geometria di simulazione.
- Costituisce un criterio logico, persistente, efficace sul quale ordinare le particelle, permettendo di accedervi in maniera coalesced.
- Permette di sfruttare la limitata memoria condivisa della GPU, se si impiegano come unità di lavoro i sottoinsiemi di spazio associati ai nodi dell'albero.
- Equilibra i carichi di lavoro: l'opportuna scelta della soglia di adattività permette di associare ad ogni nodo una comparabile quantità di elementi da simulare, e quindi di lavoro da svolgere.
- Riduce la divergenza dei warp: è ragionevole aspettarsi che, se gli spazi racchiusi dai nodi sono sufficientemente contenuti, le particelle avranno comportamenti simili e non costringeranno i kernel a seguire differenti percorsi di esecuzione.
- Aiutano il funzionamento delle cache: per dispositivi di Compute Capability 2.x e superiori gli accessi alla memoria globale sono coadiuvati dalla cache L2, che lavorerà molto quando non sarà possibile sfruttare la shared memory; all'interno di un singolo nodo dell'octree, le particelle accederanno sempre alle stesse celle, aumentando il riutilizzo dei dati nella cache.
- Facilita l'estensione dell'algoritmo ad un contesto multi-GPU.

- Predispone l'algoritmo per l'inclusione di moduli per la simulazione fisica della collisione tra particelle: il primo stadio di un algoritmo per il rilevamento delle collisioni, chiamato *broad phase*, solitamente prevede il sorting delle particelle in nodi di un octree, cosa che nel nostro caso si sarebbe già effettuata, sebbene con altri scopi.

L'esecuzione pilotata da un octree ha pertanto la possibilità di regolarizzare anche operazioni fortemente non deterministiche come un PIC non strutturato. Per ora, comunque, non ci occupiamo di dotare il codice di componenti collisionali o di estenderlo a più di un processore grafico. Queste possibilità, per quanto attraenti, rivestono un'importanza secondaria rispetto al riprodurre i comportamenti fisici fondamentali del plasma. Purtroppo al momento ci troviamo costretti a rinunciare anche alla possiblità di servirci della shared memory. L'uso di questo strumento hardware, molto desiderabile e, come si è visto, ampiamente usato, è andato incontro a difficoltà non facilmente superabili, che verranno approfondite nel Capitolo 8.

A questo punto, è necessario stabilire quale sia il parametro che guidi la adattività dell'octree. È possibile scegliere la densità di celle della griglia non strutturata, o la densità di particelle. Notiamo che, indipendentemente da questa scelta, le particelle possono sempre essere riordinate in buffer lineari e con una certa corrispondenza spaziale. Aggiungiamo che la densità di particelle cambia durante la simulazione, portando a dover ricostruire l'octree ad ogni iterazione. Invece, usando come riferimento la mesh, che rimane invariata, l'octree può essere costruito una volta sola in fase di inizializzazione, e porterebbe anche un maggiore beneficio, data la minore località spaziale della mesh. Considerando tutte queste ragioni, la scelta della soglia di adattività cade inevitabilmente sul numero di tetraedri interessati da un nodo dell'octree.

Impostato l'albero, tramite un'operazione di attraversamento (in gergo informatico *tree traversal*), le particelle possono risalire al nodo foglia a cui appartengono. Se i nodi dell'octree vengono numerati con il criterio della semplice e già accennata Z-order curve (pensata proprio per esprimere dati multidimensionali in ordine lineare mantenendone la località), allora le particelle possono essere riordinate in base al nodo di appartenenza (non è importante che le particelle abbiano un ordine definito all'interno dello stesso nodo). Dopo il sorting, i buffer delle particelle saranno costituiti da tronconi contigui, correlabili ai nodi foglia dell'octree, e accessibili in maniera coalesced.

Come si è probabilmente intuito, l'idea è quella di eseguire gli stadi PIC associando un blocco di thread ad ogni nodo foglia dell'albero, correlando unità esecutiva di parallelismo con unità di partizionamento spaziale. Ovviamente un nodo facilmente conterrà più particelle di quanti thread siano presenti in un blocco, che dovrà pertanto avvalersi di un ciclo `for` per operare su tutte la particelle ad esso assegnate.

Per sfruttare al meglio il parallelismo del dispositivo, quando sia necessario eseguire le funzioni che operino sulle particelle ci serviamo anche di diversi CUDA Streams per eseguire indipendentemente un kernel di integrazione per ogni specie fisica, aprendo la strada anche ad altri convenienti accorgimenti, alcuni dei quali descritti in Sezione 6.1.

Possiamo finalmente delineare il nostro algoritmo e illustrare come si inserisca nella procedura prevista da F3MPIC:

Algoritmo PIC su GPU:

1. *Inizializzazione F3MPIC;*
2. *Trasferimento su GPU della mesh e di altri dati necessari alla simulazione;*
3. *Partizionamento del dominio e creazione dell'octree;*
4. *Deposizione di carica iniziale;*

5. *Soluzione delle equazioni di campo tramite GetDP;*
6. *Aggiornamento dei campi EM;*
7. *Trasferimento su GPU di specie e particelle;*
8. *Tree traversal;*
9. *Sorting particelle;*
10. *Interpolazione campi EM ed avanzamento particelle;*
11. *Tracciamento particelle;*
12. *Deposizione di carica;*
13. *Trasferimento su Host di densità di carica, specie e particelle;*
14. *Generazione di nuove particelle;*
15. *Ripetere passi da 5 a 14 per il numero di iterazioni previsto.*

Uno dei nostri obiettivi finali è anche quello di realizzare un software il meno invasivo possibile per il codice originale, ed il più intuitivo e trasparente possibile per un utilizzatore finale, che non necessariamente conosce le GPU ed il loro funzionamento. L'interfaccia esterna è stata progettata per sottolineare l'intento di dare una possiblità aggiuntiva a F3MPIC, invece di forzare un cambio totale di paradigma. Per questa sua caratteristica, il nostro codice è stato battezzato *F3MPIC GPU Modules*, abbreviato in *FGM*. Questi moduli software vengono consegnati sotto forma di una libreria condivisa (e linkabile dalla build di F3MPIC senza invocare il compilatore CUDA) e del corrispondente un file di intestazione, che descrive i prototipi dell'interfaccia esterna ed i nuovi tipi di dati introdotti da FGM. Le funzioni che compongono l'interfaccia sono:

```
   int fgm_init(fgm_container *fgmc, Node *n, Triangle *tri,
Tetrahedron *tet, Species *s, int verbose);

   int fgm_simstep(fgm_container *fgmc, Node *n, Triangle *tri,
Tetrahedron *tet, Species *s, int verbose);

   void fgm_close(fgm_container *fgmc);
```

`fgm_container` è una struttura che funge da contenitore per puntatori e dati necessari a FGM, mentre `Node`, `Triangle`, `Tetrahedron` e `Species` sono le medesime strutture già esistenti in F3MPIC per rappresentare le entità di simulazione; l'ultimo parametro in ingresso, ove presente, è una flag per controllare la verbosità delle funzioni. Le funzioni di inizializzazione e simulazione ritornano un intero che indica il successo o meno della loro esecuzione; sta alla funzione chiamante controllare questo output e intraprendere opportune azioni in caso di fallimento.

Per abilitare FGM e fruire della eventuale presenza di una GPU è sufficiente chiamare `fgm_init()` dopo l'inizializzazione di F3MPIC e sostituire nel ciclo di simulazione le funzioni di interpolazione/integrazione, tracking e deposizione con un'unica invocazione a `fgm_simstep()`.

Con riferimento all'algoritmo precedentemente delineato, `fgm_init()` comprende i passi dal 2 al 3, mentre `fgm_simstep()` racchiude i passi dal 7 al 13. `fgm_close()` chiude la libreria e libera qualsiasi porzione di memoria allocata per il suo funzionamento.

Descriviamo ora nel dettaglio i componenti di FGM.

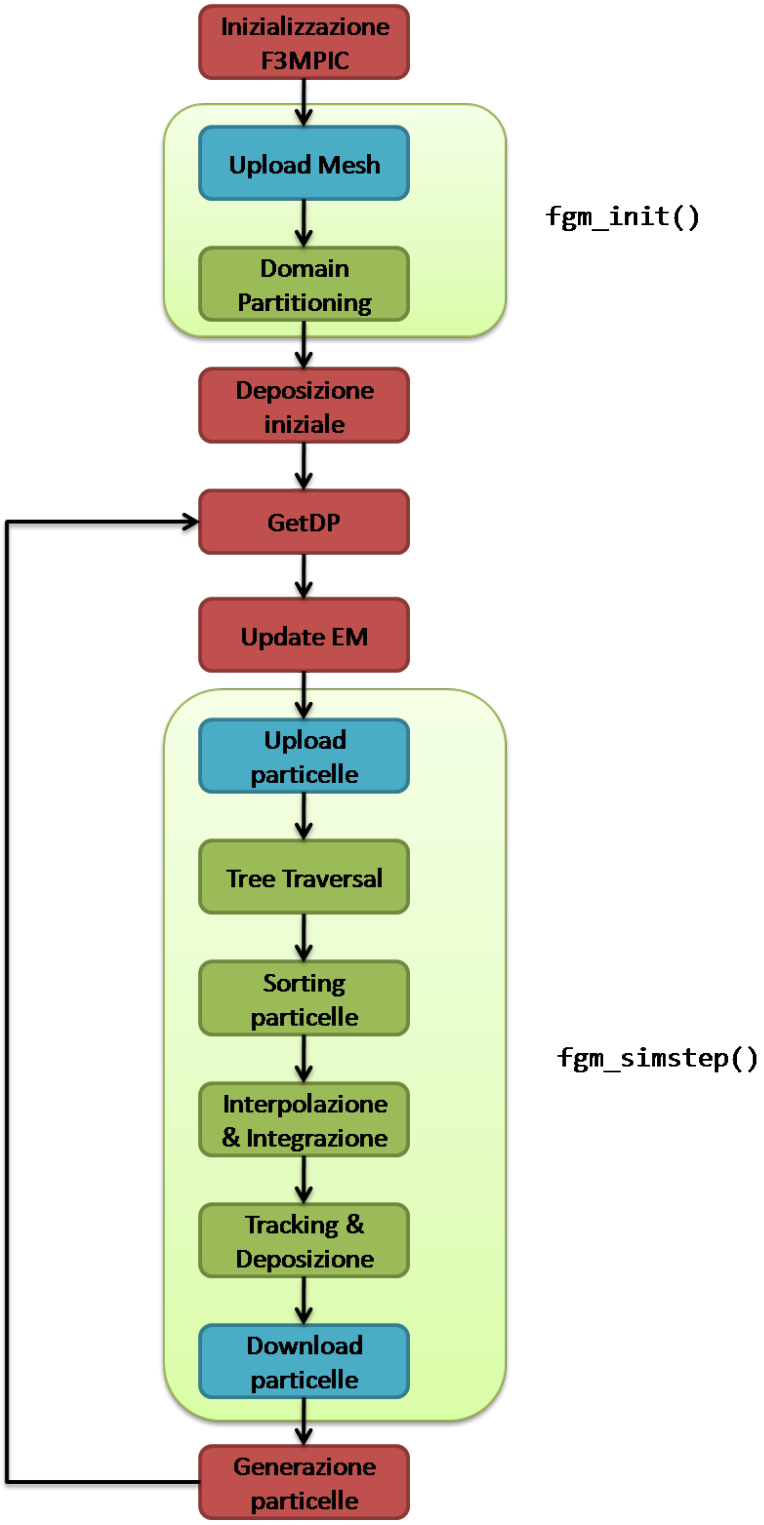


Figura 6.2: Schema generale del nuovo algoritmo PIC su GPU. In rosso sono indicati i passi eseguiti sulla CPU, in azzurro le operazioni di trasferimento dati, in verde i kernel eseguiti sulla GPU. I riquadri traslucidi illustrano come i passi sono ripartiti nelle funzioni di FGM

6.1 Considerazioni sull'ordinamento e la gestione della memoria

Per quanto riguarda l'ordinamento della memoria, valgono le considerazioni ed i risultati della relazione [12]. Alla luce di questi fatti, si sono realizzate per tutte le strutture più importanti di F3MPIC delle controparti organizzate come strutture di array, e precedute dal prefisso `Dev_`, ad indicare la loro finalità di impiego sul dispositivo GPU.

Tutte le proprietà nelle strutture di nodi, triangoli e tetraedri, anche quelle rappresentate da array, sono state sostituite con puntatori. Per proprieta di N componenti si e' scelto di usare un buffer unico, grande N volte gli elementi, con un padding a N della proprietà interessata. Ciò è stato fatto per riferirsi alle varie proprietà senza troppi problemi in caso di accesso programmatico e per sfruttare meglio il caching del dispositivo: quando viene richiesta una proprietà dalle molteplici componenti (come ad esempio una grandezza vettoriale), e' molto probabile che ne servano più componenti, se non tutti. In questa prospettiva, i componenti sequenziali sono tutti caricati in cache, cosa che non avverrebbe con i componenti divisi in buffer differenti. Questa considerazione acquista maggior valore in luce del fatto che le entità geometriche (tetraedri, nodi,triangoli) non sono ordinate spazialmente, come già osservato. L'accesso ai loro valori sara' dunque sempre disordinato, e pertanto la possibilità di attingere più dati dalla cache risulta preferibile a suddividere i singoli componenti in buffer differenti, l'accesso ai quali risulterebbe in un maggiore spreco di banda di memoria.

Per poter essere caricati sul dispositivo sotto forma di strutture di array, i dati originariamente creati da F3MPIC come array di strutture devono essere convertiti. Per semplificare la manipolazione dei dati e la loro gestione una volta allocati sulla memoria della GPU, ci serviamo di una copia di ciascuna struttura creata, che però risiede stabilmente nella memoria del sistema ospite. Queste copie sull'host, sebbene possano sembrare superflue, in realtà sono enormemente utili, principalmente perchè i puntatori a spazi di memoria sul dispositivo non sono dereferenziabili in codice eseguito sull'host. Sarebbe dunque impossibile allocare una struttura di array sul device e poi riempirla con dei buffer lineari dall'host, poichè quest'ultimo non riuscirebbe a risalire all'ubicazione delle regioni dove operare i trasferimenti. Avendo a disposizione le copie dal lato host, possiamo allocare uno alla volta le porzioni lineari di memoria sul device, e salvare il puntatore (l'unica limitazione è non dereferenziarlo, ma il suo valore è chiaramente visibile) nella struttura host. Una volta allocati e trasferiti tutti i campi necessari, è sufficiente copiare la struttura stessa (che altro non è che una lista di puntatori), per rendere noti i nuovi spazi di memoria anche ai kernel, che operano esclusivamente sulla GPU.

Le strutture sull'host non vengono liberate dopo i caricamenti, in quanto continuano a fungere da contenitori sicuri per i puntatori sul dispositivo, che sono quindi prontamente accessibili senza bisogno di essere ri-copiati dalla GPU, in caso si verificasse la necessità del passaggio a qualche funzione o di procedere alla completa pulizia del dispositivo. Senza la disponibilità dei puntatori nelle strutture host, nel caso qualcosa comprometta l'esecuzione del programma, non sarebbe possibile deallocare la memoria riservata per la simulazione, e la GPU si troverebbe a funzionare con ampie zone di memoria "fantasma", che non è possibile utilizzare nè liberare (fenomeno del *memory leaking*).

I singoli elementi della mesh di F3MPIC (nodi, triangoli, tetraedri) sono anche correlati tra loro da dei puntatori reciproci, che indicano per esempio quali siano i nodi che formano i vertici di un tetraedro, o quali triangoli ne formino le facce. A causa della non dereferenziabilità dei puntatori a device, questo approccio deve essere cambiato. Sarebbe infatti impossibile risalire ad indirizzi di entità precedentemente allocate. Nelle strutture di FGM questi puntatori reciproci sono stati sostituiti con gli indici dell'elemento a cui fanno riferimento, ricavati tramite

operazioni di aritmetica dei puntatori a partire dai dati originari forniti da F3MPIC. Questi indici vengono poi raggruppati in array lineari di interi e copiati sulla GPU come un qualsiasi altro dato.

Si è anche osservato che alcuni parametri di simulazione, come la durata del passo temporale, e alcune caratteristiche fisiche proprie di ogni specie (per esempio carica e massa), rimangono invariate durante tutta la simulazione (pertanto possono essere considerate dati a sola lettura), ed inoltre sono richieste da tutti i thread nelle medesime fasi dell'integrazione. Queste proprietà le rendono eccellenti candidate per essere caricate nella memoria costante del dispositivo e sfruttandone al meglio le capacità di caching e broadcasting, e si sposano benissimo con la strategia di lanciare un kernel per ogni singola specie in stream differenti: l'indice dello stream coincide con il valore identificativo della specie fisica, e si può passare al kernel lo stesso contatore usato per chiamare in sequenza gli stream.

6.2 Partizionamento del dominio di calcolo

Per poter procedere alla suddivisione della griglia nei nodi dell'octree, è necessaria una procedura per determinare quali tetraedri effettivamente intersechino lo spazio di forma cubica (da ora in poi chiamato *box* per semplicità) associato ad un nodo dell'albero. Impieghiamo una metodologia dovuta a Ratschek e Rokne [13], che fa uso di una ingegnosa quanto elegante combinazione di aritmetica degli intervalli e coordinate baricentriche.

Introduciamo brevemente questi concetti per maggior chiarezza.

6.2.1 Artimetica degli intervalli

La matematica degli intervalli è un metodo sviluppato dai matematici fin dagli Anni '50 come approccio per contenere errori di arrotondamento e di misurazione nei calcoli matematici, definendo dei metodi numerici che diano risultati affidabili. Molto semplicemente, la matematica degli intervalli rappresenta ogni valore come un intervallo di possibilità. L'argomento è molto vasto, e per i nostri scopi è sufficiente limitarsi all'aritmetica di base.

Le operazioni elementari dell'aritmetica degli intervalli sono, per due intervalli $[x_1, x_2], [y_1, y_2] \in \mathbb{R}^2$,

$$\begin{array}{ll} \textit{Addizione}: & [x_1,x_2]+[y_1,y_2]=[x_1+y_1,x_2+y_2]; \\ \textit{Sottrazione}: & [x_1,x_2]-[y_1,y_2]=[x_1-y_2,x_2-y_1]; \\ \textit{Moltiplicazione}: & [x_1,x_2]\cdot[y_1,y_2]= \\ & [min(x_1y_1,x_1y_2,x_2y_1,x_2y_2),max(x_1y_1,x_1y_2,x_2y_1,x_2y_2)]; \\ \textit{Divisione}: & [x_1,x_2]/[y_1,y_2]=[x_1,x_2]\cdot(1/[y_1,y_2]), \\ & \textit{dove} \quad 1/[y_1,y_2]=[1/y_1,1/y_2] \quad \textit{se} \quad 0\notin[y_1,y_2]. \end{array} \tag{6.1}$$

È prevista una particolare casistica per trattare la divisione per un intervallo che includa lo 0.

Dal momento che un numero reale $r \in \mathbb{R}$ può essere interpretato come l'intervallo $[r, r]$, intervalli e numeri reali possono essere combinati liberamente e facilmente.

6.2.2 Coordinate baricentriche

In geometria, il sistema di coordinate baricentrico è definito come un sistema di coordinate nel quale la posizione del punto di un simplesso (un triangolo in due dimensioni, un tetraedro in tre, e così) via) è specificata come il centro di massa, o baricentro, di un sistema di masse poste ai vertici. Le coordinate baricentriche sono spesso usate per esprimere la posizione relativa tra un punto ed un simplesso.

Consideriamo ora che i quattro punti $a, b, c, d \in \mathbb{R}^3$ formino i vertici di un tetraedro $T \subseteq \mathbb{R}^3$. Le coordinate baricentriche $\gamma_i(p)$ di un punto $p \in \mathbb{R}^3$ rispetto al tetraedro sono calcolate come

$$\begin{cases} \gamma_1(p) &= \dfrac{vol(p,b,c,d)}{vol(a,b,c,d)} \\ \gamma_2(p) &= \dfrac{vol(a,p,c,d)}{vol(a,b,c,d)} \\ \gamma_3(p) &= \dfrac{vol(a,b,p,d)}{vol(a,b,c,d)} \\ \gamma_4(p) &= \dfrac{vol(a,b,c,p)}{vol(a,b,c,d)} \end{cases} \tag{6.2}$$

dove in generale

$$vol(r,s,t,u) = \frac{1}{6}\begin{vmatrix} r_1 & r_2 & r_3 & 1 \\ s_1 & s_2 & s_3 & 1 \\ t_1 & t_2 & t_3 & 1 \\ u_1 & u_2 & u_3 & 1 \end{vmatrix} \tag{6.3}$$

con $r, s, t, u \in \mathbb{R}^3$. Anche se questa definizione è completa si fa notare che $vol(r, s, t, u)$ è il volume del tetraedro orientato con vertici r, s, t, u (in quest'ordine), il quale può essere degenere.

Le coordinate baricentriche hanno alcune proprietà molto interessanti. Evidenziamo le seguenti:

$$p \in T \quad se \quad 0 \leq \gamma_i(p) \leq 1, \quad i = 1, \ldots, 4; \tag{6.4}$$

$$\sum_{i=1}^{4} \gamma_i(p) = 1, \quad per \quad p \in \mathbb{R}^3. \tag{6.5}$$

6.2.3 Intersezione tra un box e un tetraedro

Enunciamo prima di tutto un principio che tornerà utile in seguito.

Sia $\mathbb{I}$ l'insieme degli intervalli reali compatti. Se $X \in \mathbb{I}^3$, allora il range di una funzione f su X è denotato da $\overline{f}(X) = f(x) : x \in X$. L'estensione naturale ad intervallo di f si denota con $f(X)$ ed è definita come quell'espressione a intervalli che emerge se le variabili x_1, x_2, x_3 in $f(x_1, x_2, x_3)$ sono sostituite rispettivamente con gli intervalli X_1, X_2, X_3. In generale, si ha solo la relazione

$$f(X) \supseteq \overline{f}(X). \tag{6.6}$$

Principio: Se l'espressione $f(x_1, x_2, x_3)$ è un'espressione aritmetica, cioè consente solo l'esecuzione delle quattro operazioni aritmetiche, e se ogni variabile interviene al più una volta, allora il range e l'estensione naturale ad intervallo coincidono:

$$f(X) = \overline{f}(X). \tag{6.7}$$

Il valore del principio sta nel fatto che il box completo B, ovvero l'insieme di tutti i suoi punti, quando descritto da coordinate baricentriche, può essere interpretato esattamente come *un'unica* espressione di aritmetica degli intervalli.

Ora si consideri nuovamente un tetraedro $T \subseteq \mathbb{R}^3$, un punto $p \in \mathbb{R}^3$ e definiamo un un box B come costruito da tre intervalli: $B = B_1 \times B_2 \times B_3$. Più rigorosamente, potremmo scrivere $B = (B_1, B_2, B_3),\ B_i \in \mathbb{I},\ i = 1, 2, 3$. Allora

$$B \subseteq T \quad se \quad 0 \leq \gamma_i(p) \leq 1, \quad i = 1, \ldots, 4, \quad \forall\, p \in B. \tag{6.8}$$

Si faccia bene attenzione che, essendo B definito da intervalli, anche le sue coordinate baricentriche rispetto a T sono intervalli! L'espressione $\forall\, p \in B$ significa che tutti i punti nel range di $\gamma_i(p)$ su B, definito come $\overline{\gamma}_i(B) = \{\gamma_i(p) |\ p \in B\}$, devono verificare la condizione (6.8). In altre parole, B è contenuto in T se tutte le coordinate baricentriche giacciono nell'intervallo $[0, 1]$.

Per semplicità, prendendo $i = 1$ (gli altri casi sono analoghi) si ha

$$\gamma_1(B) = \frac{vol(B, b, c, d)}{vol(a, b, c, d,)} \supseteq \overline{\gamma}_i(B), \tag{6.9}$$

usando l'aritmetica degli intervalli per l'estensione naturale ad intervallo di γ_1 su B. Dal momento che il denominatore è uno scalare, dobbiamo considerare solo il numeratore per ottenere gli estremi del range, e abbiamo

$$vol(B, b, c, d) = \frac{1}{6} \begin{vmatrix} B_1 & B_2 & B_3 & 1 \\ b_1 & b_2 & b_3 & 1 \\ c_1 & c_2 & c_3 & 1 \\ d_1 & d_2 & d_3 & 1 \end{vmatrix} =$$

$$= \left.\begin{array}{l} \{(B_1 - d_1)((b_2 - d_2)(c_3 - d_3) - (b_3 - d_3)(c_2 - d_2)) \\ -(B_2 - d_2)((b_1 - d_1)(c_3 - d_3) - (b_3 - d_3)(c_1 - d_1)) \\ +(B_3 - d_3)((b_1 - d_1)(c_2 - d_2) - (b_2 - d_2)(c_1 - d_1))\}/6 \end{array}\right\}; \tag{6.10}$$

possiamo applicare il principio prima menzionato, poichè ogni variabile intervallo compare solo una volta. Ciò significa che per questa particolare espressione vale

$$\overline{vol}(B, b, c, d) = vol(B, b, c, d). \tag{6.11}$$

Valutando i determinanti secondo la (6.10) ed usando il valore scalare $vol(a, b, c, d)$, segue che i vari $\gamma_i(B),\ i = 1, \ldots, 4$ espressi dalla (6.9) possono essere impiegati in una procedura per ottenere una risposta definita riguardo la relazione fra il box ed il tetraedro.

A questo punto, riportiamo l'algoritmo da noi usato, che è leggermente più semplice della procedura di Ratschek e Rokne, la quale è in grado di rilevare anche casi di sovrapposizione e inclusione completa. Tuttavia, alcuni passi sono di difficile parallelizzazione e non sono di alcun aiuto verso i nostri scopi, pertanto si è deciso di trascurarli.

Algoritmo per rilevare l'intersezione tra un box B ed un tetraedro T:

1. *Controllare il numero di vertici di T giacenti in B;*
   se $\neq 0$ allora B e T si intersecano.

2. *Calcolare H, il box di inviluppo isotetico di T;*
   calcolare $B_H = H \cap B$.

3. *Se $B_H = \emptyset$, allora B e T sono disgiunti.*

4. *Per* $i = 1, \ldots, 4$,
   (i) calcolare l'intervallo di coordinate baricentriche $\gamma_i(B_H)$*;*
   (ii) se $\gamma_i(B_H) < 0$ *oppure* $\gamma_i(B_H) > 1$*, allora* B *e* T *sono disgiunti.*

5. *Se non è stata presa alcuna decisione nei passi precedenti, allora* B *e* T *si intersecano.*

L'algoritmo fa uso del box di inviluppo isotetico (cioè allineato agli assi coordinati) H del tetraedro perchè si dimostra che la sua intersezione B_H con il box di riferimento B conserva le stesse proprietà di intersezione con T. Operando con B_H c'è una maggiore possibilità che l'algoritmo restituisca un risultato precocemente.

6.2.4 Costruzione dell'octree

Con gli strumenti teorici appena introdotti, possiamo implementare il codice che partizionerà il dominio, costruendo via via l'octree.

L'albero in memoria è costituito da un array di apposite strutture, ognuna delle quali rappresenta un nodo. I nodi sono identificati da due valori: il livello gerarchico (che si può anche chiamare profondità) di appartenenza e la loro posizione tra i nodi dello stesso livello. Si è scelto di disporre progressivamente in memoria i differenti livelli di profondità, mentre i nodi appartenenti alla medesima profondità sono contigui: si avranno dunque prima il nodo radice, poi gli 8 nodi del primo livello, poi i 64 nodi del secondo livello, e così via. All'interno dello stesso livello, i nodi sono numerati seguendo il criterio di una curva Z.

Seguendo questi criteri, è possibile determinare l'indirizzo adr_n di un dato nodo n nell'intero albero usando la seguente espressione:

$$adr_n = N_{abv}(lvl) + 8 \times (ID_g) + ID_c \tag{6.12}$$

dove lvl è il livello del nodo n, ID_g è l'identificativo del suo nodo genitore internamente al livello del genitore, e ID_c è l'identificativo di n tra i suoi "fratelli" (intesi come i nodi derivati dallo stesso genitore), anch'esso determinato con la regola della curva Z. N_{abv} è una funzione che restituisce il numero di nodi totali sopra al livello lvl, ipotizzando che livelli più alti abbiano un identificativo inferiore. Come tale, N_{abv} non è altro che la somma di tutte le potenze intere di 8 da 0 fino a $lvl - 1$. Per esempio,

$$N_{abv}(5) = 8^4 + 8^3 + 8^2 + 8^1 + 8^0 = 4681. \tag{6.13}$$

La costruzione comincia in codice host, dove viene fissato il massimo livello di profondità dell'octree: si preferisce allocare un numero noto a priori di nodi e, qualora la mesh fosse troppo fitta, è meglio ritornare un messaggio di errore che esaurire la memoria continuando ad espandere l'albero. Attualmente si è impostata una profondità massima pari a 5, che equivale a 32768 nodi all'ultimo livello ed un totale di 37449 nodi. Si procede determinando il punto centrale della mesh a partire dai suoi estremi (informazione già fornita da F3MPIC) e, con queste informazioni, si configura il nodo radice sull'host. Viene quindi allocato l'intero albero sul device e copiato il nodo radice.

Il raffinamento adattivo dell'albero è eseguito completamente dalla GPU sfruttando la tecnologia di parallelismo dinamico descritta in Sezione 4.3.1. Siccome c'è un limite alla profondità di griglie CUDA lanciabili dal device runtime, impostiamo la massima profondità dell'albero come massimo livello ammesso per il Dymanic Parallelism.

All'interno del kernel che opera il partizionamento del dominio, un blocco corrisponde ad un singolo nodo dell'octree. I thread del blocco verificano ciascuno se un tetraedro interseca il box di guardia del nodo, servendosi della procedura precedentemente descritta; in caso di esito positivo, si aumenta atomicamente un contatore. Una volta esauriti i tetraedri, se il valore della

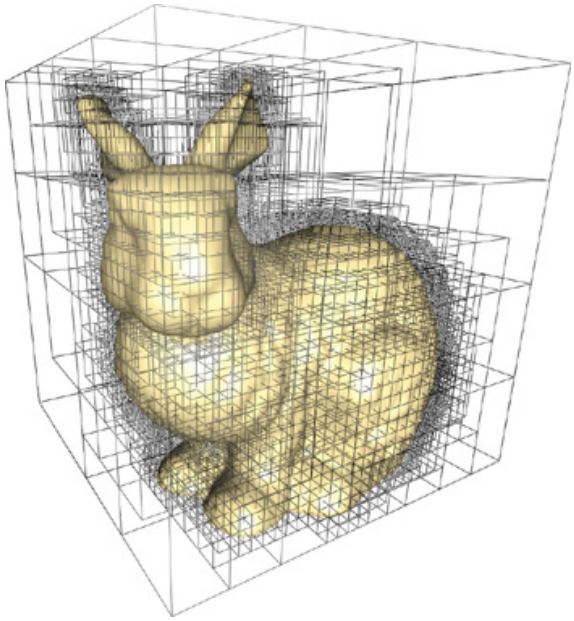


Figura 6.3: Un octree costruito adattivamente attorno ad un modello

somma e‘ minore o uguale al numero di tetraedri stabilito come soglia, il nodo viene identificato come foglia, i suoi dati vengono salvati nei buffer che indicano i nodi da elaborare per gli stadi PIC ed il kernel ritorna correttamente alla funzione chiamante. Al contrario, se la somma eccede la soglia, il controllo dei tetraedri viene interrotto, ed il primo thread del blocco verifica il livello del nodo a cui appartiene. In caso sia minore del livello massimo consentito, il thread trova gli indirizzi dei nodi figli con la formula (6.12), ne configura i box e lancia una nuova griglia di calcolo con esattamente 8 blocchi, ognuno dei quali prenderà in consegna un nodo figlio. Se invece il livello è pari al massimo, non è consentito il lancio di ulteriori blocchi, viene restituito un messaggio di errore e l'esecuzione ritorna alla funzione chiamante.

Il kernel procede ricorsivamente, chiamando ripetutamente sé stesso man mano che si addentra in ciascun ramo dell'albero.

6.3 Riordinamento delle particelle

Con l'octree completamente definito, è necessario abbinare ogni particella al nodo foglia in cui si trova, copiando l'indice globale di tale nodo nella proprietà `box_id`, di cui ogni particella è appositamente predisposta. Il compito spetta al kernel di tree traversal, i cui thread, uno per ciascuna particella allocata, percorrono l'albero fino a giungere all'estermità della loro particella. L'attraversamento viene effettuato con metodologia *breadth-first*, ovvero controllando prima i nodi fratelli giacenti su uno stesso livello, individuando il ramo corretto per la particella e proseguendo al livello successivo. È importante comunque notare che F3MPIC in ogni momento utilizza nella simulazione solo una parte (comunque preponderante) delle particelle allocate, per gestire meno aggressivamente la memoria in corrispondenza dell'uscita o della introduzione di particelle nel dominio. FGM mantiene questa caratteristica, attribuendo alle particelle segnalate come inutilizzate un valore di `box_id` superiore a qualunque identificativo possibile per i nodi dell'octree. In questo modo, al momento del sort le particelle inutilizzate verranno tutte raggruppate nelle code dei buffer, assecondando anche il funzionamento di F3MPIC, che cerca particelle libere da reintrodurre nel dominio partendo dal termine delle allocazioni.

Terminato il traversal, si procede al sorting delle particelle in base al loro `box_id`. L'algoritmo di sorting che riordina delle variabili non secondo il loro valore ma seguendo un dato ad esse associato (in gergo detto *chiave*) prende il nome di *radix sort*. Per nostra fortuna, questo è uno dei tipi di sorting più affermati, potenti e ricercati per le GPU, e dunque facilmente reperibile in risorse fornite da terzi. Seguendo le esperienze di Payne [8], anche noi ci affidiamo all'implementazione del radix sort inclusa nella libreria Thrust, molto facile da utilizzare ma al tempo stesso la più performante per casi non specifici. L'unica azione propedeutica all'invocazione del sort è l'includere i puntatori grezzi usati da FGM all'interno degli appositi contenitori previsti da Thrust per i puntatori al device.

Thrust tuttavia non si occupa dello spostamento di tutti i campi delle strutture: il suo esito è semplicemente un vettore di interi che riporta gli indici delle particelle prima del sort, ma nell'ordine che invece dovrebbero avere. Provvediamo dunque a riordinare le informazioni di interesse (denominate *particle payload* dal Payne) con opportuni kernel. In questa circostanza, torna utile anche un altro punto di forza del sorting basato su Thrust: è la strategia che richiede il minor spreco di memoria per riordinare i payload.

Infatti, a differenza di tecniche che richiedono anche una replica completa di tutte le strutture relative alle particelle, nel nostro caso è sufficiente avere a disposizione un singolo buffer libero (soprannominato *sorting buffer*) per ognuno dei tipi di dato da riordinare, ovviamente di estensione pari al numero di particelle allocate. Agendo su un attributo alla volta, il kernel di spostamento del payload legge il vecchio indice di una particella, e sposta l'attributo nel sorting buffer, scrivendolo nella posizione prevista dal radix sort. Dopo questa operazione, i puntatori ai buffer vengono scambiati: quello che prima era uno spazio ausiliario, ora ospita dei dati utili, mentre il buffer di origine (che contiene ancora dati precendenti al sort) è superfluo e può essere usato come nuovo sorting buffer. Agendo su tutte gli attributi delle particelle, i buffer vengono usati ciclicamente, con la sola condizione che ce ne sia sempre uno disponibile ad accogliere la scrittura di dati riordinati.

Quando tutte le particelle sono state disposte nella nuova configurazione, si aggiornano i dati relativi alle particelle contenute in ciascun nodo foglia dell'octree.

Un kernel lancia un thread per ogni particella interna al dominio di simulazione, ed i thread leggono 3 valori di `box_id`: quello della loro particella e delle 2 particelle adiacenti. Se il `box_id` della particella corrente è diverso dal precedente o dal successivo, significa che la particella costituisce, rispettivamente, il primo o l'ultimo elemento dell'intervallo di particelle da associare al suo nodo foglia. Questa informazione viene salvata nel nodo e sarà poi letta dai kernel di interpolazione, integrazione e deposizione per conoscere gli indici delle particelle a cui accedere coi loro cicli interni.

6.4 Interpolazione ed avanzamento delle particelle

Trasporre l'interpolazione dei campi e l'integratore leapfrog-Vay in un codice adatto all'esecuzione su GPU si dimostra alquanto semplice: entrambe le operazioni si presentano da subito come problemi *embarrassingly parallel*, ovvero la cui espressione parallela è pressochè immediata. Infatti ogni particella evolve il suo moto in maniera completamente indipendente rispetto a tutte le altre, e non è richiesta nessuna comunicazione nè collaborazione tra le varie unità di calcolo. I cicli `for` dell'implementazione F3MPIC possono essere "srotolati", e assegniamo ogni particella ad un singolo thread della griglia di calcolo.

Per di più, l'interfaccia di programmazione CUDA C permette di riutilizzare il medesimo codice del programma originale, salvo minimi aggiustamenti per accedere alla memoria globale

solo quando necessario per la lettura dei dati iniziali e la scrittura di quelli finali. Tutti i risultati intermedi vengono salvati su registri, che costituiscono la memoria più veloce disponibile ed in questo caso non pongono significativi problemi di occupancy.

6.5 Tracciamento delle particelle

La discretizzazione del volume di controllo attraverso una mesh non strutturata impone la necessità di implementare un algoritmo di tracciamento (o *tracking*) delle particelle, in quanto non è immediato risalire alla cella in cui sia contenuta una particella carica ad un determinato istante temporale (dato ovviamente fondamentale per l'interpolazione e la deposizione). Altrettanto utile quanto complicato, rispetto ad una discretizzazione strutturata, è conoscere le facce delle celle attraversate dalle particelle durante il loro moto.

Si è voluto cogliere l'opportunità offerta dallo sviluppo di FGM per introdurre un algoritmo di tracking completamente differente, più efficiente e geometricamente accurato, rispetto a quello adottato nella versione originale di F3MPIC. La scelta è ricaduta sull'algoritmo usato da Brünggel [14], che ha adattato alla GPU il lavoro di Macpherson et al [15] sul tracciamento della classe di oggetti `solidParticle` contenuti nel software OpenFOAM. La procedura è valida per griglie non strutturate di tipologia qualsiasi, e dunque anche per la mesh tetraedrica di F3MPIC.

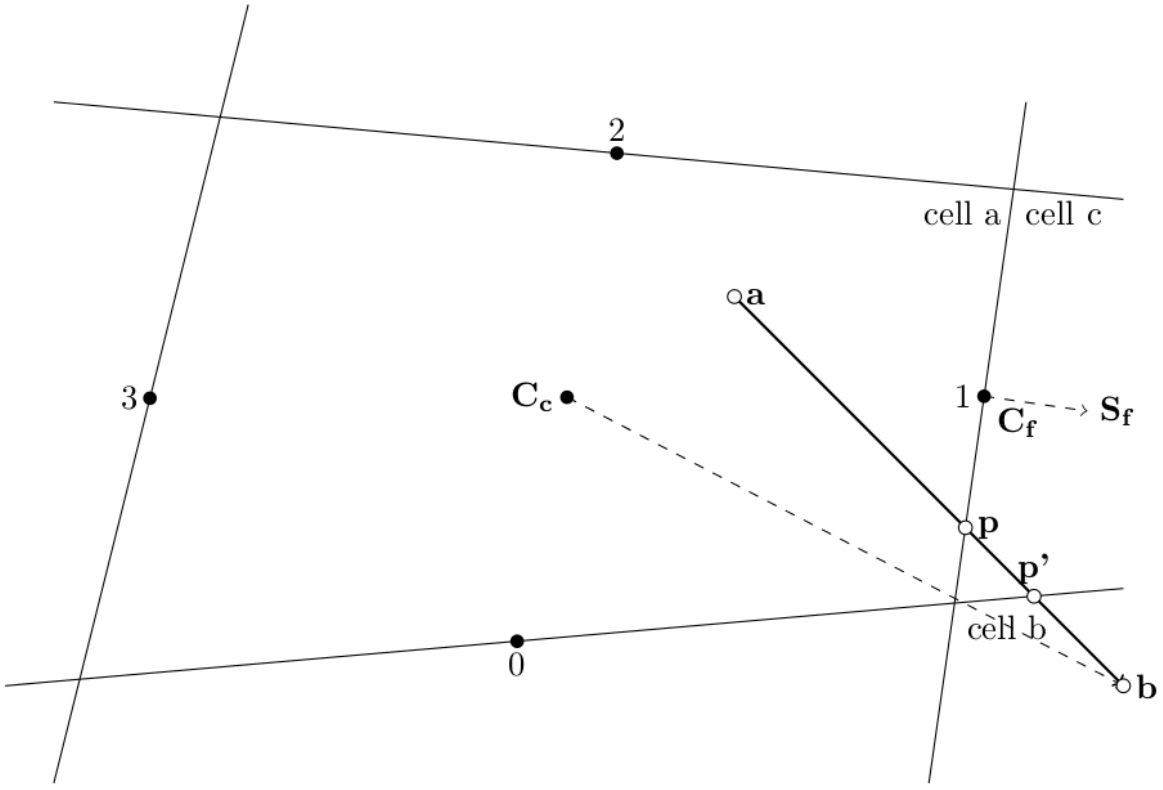


Figura 6.4: Esempio dell'algoritmo di tracking per un caso bidimensionale

La Figura 6.4 illustra l'algoritmo di tracking durante un passo temporale. Una particella, inizialmente localizzata in **a**, si muove alla posizione **b** cambiando due volte cella nei punti **p**

e $\mathbf{p}'$. Il punto nel quale la particella attraversa la prima faccia è calcolabile con la seguente equazione:

$$\mathbf{p} = \mathbf{a} + \lambda_a \cdot (\mathbf{b} - \mathbf{a}), \tag{6.14}$$

dove λ_a indica la frazione del vettore spostamento che la particella percorre fino ad intersecare la faccia. Nella (6.14), λ_a e $\mathbf{p}$ sono incognite e si conoscono solo gli estremi del movimento della particella. Il vettore dal centroide di faccia $\mathbf{C_f}$ a $\mathbf{p}$ giace sulla faccia stessa, ed è quindi ortogonale al versore normale $\mathbf{S_f}$. Sfruttando le proprietà geometriche del prodotto scalare, possiamo scrivere una nuova relazione che ci consenta di ricavare $\mathbf{p}$:

$$(\mathbf{p} - \mathbf{C_f}) \cdot \mathbf{S_f} = 0. \tag{6.15}$$

Sostituendo ora (6.15) in (6.14) e risolvendo per λ_a otteniamo

$$\lambda_a = \frac{(\mathbf{C_f} - \mathbf{a}) \cdot \mathbf{S_f}}{(\mathbf{b} - \mathbf{a}) \cdot \mathbf{S_f}}. \tag{6.16}$$

Dal momento che $\mathbf{p}$ non è più presente nell'equazione (6.16), possiamo calcolare λ_a per ogni faccia della cella. Quella con il più piccolo $\lambda_a \in [0, 1]$ sarà la faccia attraversata dalla particella. La particella può essere mossa in $\mathbf{p}$, aggiornando il dato relativo alla sua appartenenza con la cella attigua, e ripetendo la procedura fino al raggiungimento di $\mathbf{b}$. Se nessun λ_a è interno all'intervallo $[0, 1]$, allora $\mathbf{b}$ si trova nella stessa cella di partenza.

Brünggel tuttavia nota come questo algoritmo sia inconsistente in corrispondenza dell'attraversamento delle facce: usando i piani effettivi (definiti dai versori normali) invece delle facce stesse, le celle nella mesh non riempiono più completamente lo spazio, ed è possibile perdere traccia di una particella quando questa attraversa una faccia vicino ad un vertice.

Lo stesso autore presenta una modifica che risolve questo inconveniente. Sempre con riferimento alla figura 6.4, prendiamo il centroide della cella $\mathbf{C_c}$ come riferimento per determinare quali facce la particella attraverserà. Sostituendo $\mathbf{a}$ con $\mathbf{C_c}$, la (6.16) porge

$$\lambda_c = \frac{(\mathbf{C_f} - \mathbf{C_c}) \cdot \mathbf{S_f}}{(\mathbf{b} - \mathbf{C_c}) \cdot \mathbf{S_f}}. \tag{6.17}$$

La linea da $\mathbf{C_c}$ a $\mathbf{b}$ (tratteggiata in figura 6.4) attraversa i piani definiti dalle facce 1 e 0. L'espressione (6.17) dunque porge $\lambda_c \in [0, 1]$ per le facce 1 e 0. Nel caso non vi fosse alcuna faccia con $\lambda_c \in [0, 1]$, allora $\mathbf{b}$ si trova nella medesima cella di $\mathbf{a}$. È altrimenti necessario calcolare λ_a usando la (6.16) per le facce che sono attraversate dal segmento $(\mathbf{C_c}, \mathbf{b})$. Il valore più basso di λ_a determina quale faccia è stata effettivamente attraversata e la frazione di spostamento percorsa. Calcoliamo nuovamente $\mathbf{p}$ con

$$\mathbf{p} = \mathbf{a} + \lambda_m \cdot (\mathbf{b} - \mathbf{a}), \tag{6.18}$$

dove $\lambda_m = min(1, max(0, \lambda_a))$.

Riportiamo di seguito l'algoritmo completo in pseudo-codice.

Algoritmo di tracking:

mentre *la particella non ha raggiunto la sua posizione finale*
    trovare l'insieme di facce F_i *per le quali* $0 \leq \lambda_c \leq 1$
    se $F_i = \emptyset$
        muovere la particella alla posizione finale
    altrimenti

trovare la faccia $f \in F_i$ *con il minore* λ_a
muovere la particella secondo l'equazione (6.18) usando questo λ_a
impostare la cella corrente come la cella adiacente alla faccia f
fine se
fine mentre

6.6 Deposizioni di carica e comportamenti a parete

Una volta determinata la destinazione finale di una particella, F3MPIC prevede che, a seconda della sua posizione e traiettoria, essa possa trasferire alcune grandezze fisiche alla griglia spaziale, oppure essere esclusa dal volume di controllo e rimossa dalla simulazione. FGM mantiene intatte tutte le funzionalità previste dal codice originario in questa fase, tranne la simulazione delle superfici di sheath; la formulazione di questo fenomeno fisico necessita di ulteriori studi per essere tradotta efficacemente in un codice massivamente parallelo. Per il resto, come avvenuto per l'interpolazione e l'integrazione del moto, l'uso dell'API CUDA compatibile con il linguaggio C permette un riutilizzo diretto di gran parte del codice sorgente di F3MPIC, con piccoli accorgimenti per ridurre un uso eccessivo dei registri sulla GPU.

Esaminiamo la casistica implementata nel software: conclusa la fase di tracking, ad una generica particella viene assegnato un valore che indica il suo stato alla fine del passo temporale considerato. Si distinguono le seguenti situazioni:

- **La particella è interna ad una cella del dominio:** la sua carica va depositata sui nodi della cella (chiamiamo questa operazione *internal deposition*). Per determinare in che misura distribuire la quantità di carica, si moltiplica quest'ultima per i pesi associati ad ogni nodo, i quali devono essere ricalcolati, dal momento che la particella non è più nella sua posizione originaria (e probabilmente neanche nella medesima cella).

  F3MPIC adotta una formulazione lineare dei pesi, e questo è un particolare molto vantaggioso, perchè, come sottolineato in Sezione 3.3, in questo caso i pesi coincidono con le coordinate baricentriche della particella rispetto al suo tetraedro. È possibile riutilizzare le (6.2) e alcune delle funzioni già realizzate per la verifica dell'intersezione tra un box ed un tetraedro, ovviamente modificandole per operare su dati scalari (cfr. Sezione 6.2.3).

  Per sommare le cariche si opera sulla struttura dei nodi della mesh che, risiedendo nella memoria globale, è comune a tutti i thread. È indispensabile in questa situazione evitare perdite di dati dovute ad errori *read-after-write*, *write-after-read* ed altre anomalie potenzialmente causate da accessi concorrenti agli stessi indirizzi di memoria.

  A prima vista il modello CUDA non dispone di operazioni atomiche, per le quali l'hardware garantisce l'impossibilità di essere interrotte e la corretta scrittura dei dati, e si potrebbe pensare di ricorrere alla tecnica di "thread tagging"proposta da Kong [6]. Tuttavia, la Guida di Programmazione CUDA [17] fa notare che è possibile realizzare qualsiasi operazione atomica attraverso la funzione `atomicCAS()` (atomic Compare and Swap). È addirittura possibile realizzare atomiche in doppia precisione, anche se `atomicCAS()` opera su interi, servendosi di funzioni intrinseche che interpretano una sequenza di 64 bit come un intero `long` oppure come un numero in virgola mobile `double`. Si è verificata l'equivalenza dei due approcci e infine si è optato per la proposta della Guida CUDA.

- **La particella colpisce una superficie di parete ed appartiene ad una specie neutra:** se la superficie interessata è una parete neutra, la particella viene esclusa dalla simulazione, altrimenti si valuta la sua eventuale riflessione, dipendente dalla sua natura e

dalle caratteristiche della superficie. In caso di riflessione, il vettore velocità viene ruotato di un angolo ampio il doppio di quello formato dalla traiettoria della particella con la normale alla superficie e la particella prosegue il ciclo di tracciamento.

- **La particella colpisce una superficie Floating:** La carica della particella viene depositata nella superficie flottante con un'apposita procedura.
- **La particella colpisce una superficie Conduttore:** La particella deve essere esclusa dalla simulazione
- **La particella colpisce una superficie Emettitore:** La particella deve essere esclusa dalla simulazione
- **La particella colpisce una superficie Neutra:** La particella deve essere esclusa dalla simulazione
- **La particella colpisce una superficie Dielettrico:** deve essere valutata la riflessione della particella, a seconda della sua natura e delle caratteristiche fisiche del dielettrico. Se viene riflessa, la traiettoria viene modificata come per la riflessione dei neutri, e la particella prosegue il suo tracciamento fino alla nuova destinazione. Se non viene riflessa, la carica della particella viene depositata sui nodi della superficie dielettrico, con una tecnica analoga alla deposizione interna, ma in due dimensioni. Anche qui è necessario fare ricorso ad addizioni atomiche.

Quando tutte le deposizioni sono avvenute, la carica sui nodi viene divisa per i rispettivi covolumi, in modo da ottenere finalmente la densità di carica, necessaria per risolvere i campi elettromagnetici all'inizio dell'iterazione temporale successiva.

Capitolo 7

Validazione ed analisi dei risultati

7.1 Test Case

Per confrontare i risultati di FGM con il codice F3MPIC originale usiamo un dominio di simulazione costituito da 3 volumi cilindrici di diverso diametro ed altezza, con gli assi allineati all'asse coordinato y e disposti contiguamente lungo lo stesso asse come illustrato nelle Figure 7.1 e 7.2. Il cilindro più piccolo è disposto in mezzo agli altri in modo da creare una strozzatura. La mesh discretizza la geometria con 24552 nodi, 251147 triangoli e 120018 tetraedri.

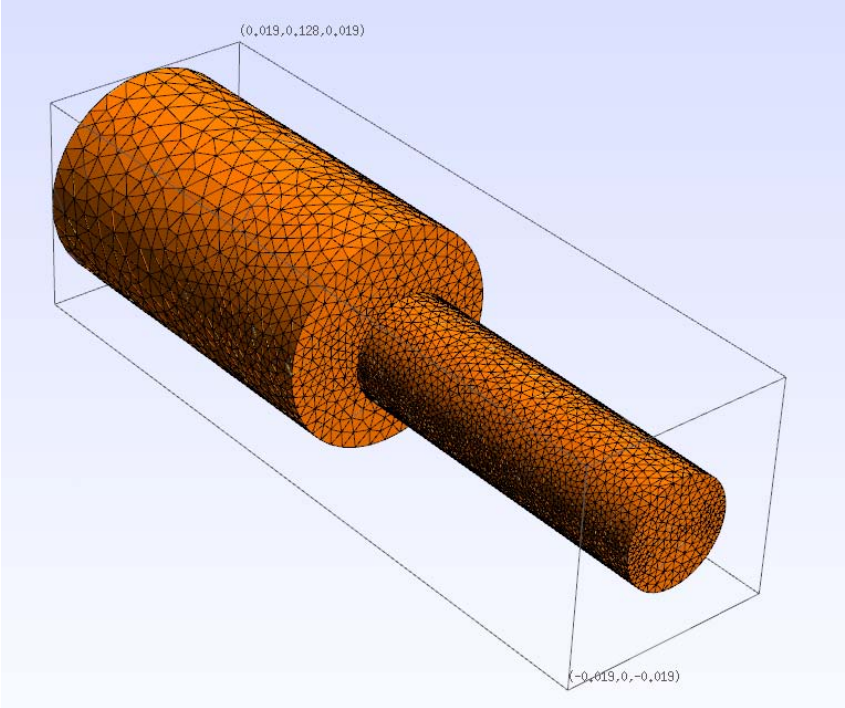


Figura 7.1: Visualizzazione del volume di controllo discretizzato. Vista in prospettiva

Tutte le superfici sono di materiale conduttore.

Il caso prevede la simulazione di 2 specie fisiche, elettroni e ioni di Argon, con le seguenti caratteristiche:

- **Elettroni:**
  - Carica $= -1.60217646 \cdot 10^{-19} C$

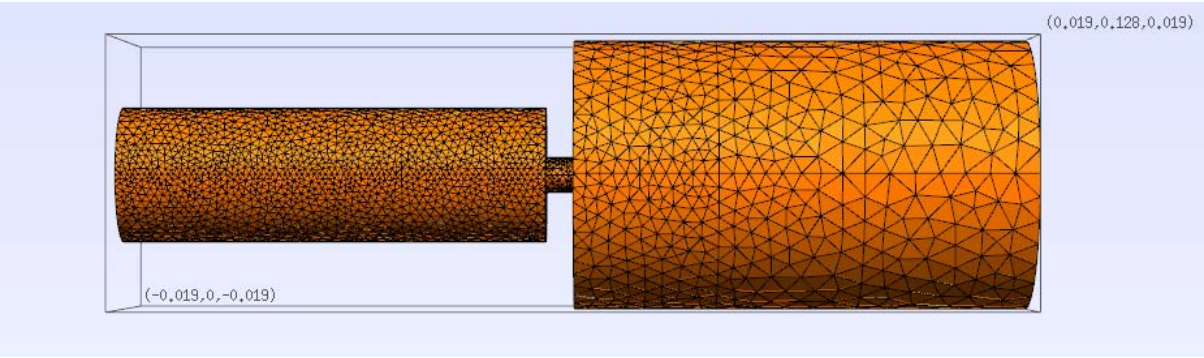


Figura 7.2: Visualizzazione del volume di controllo discretizzato. Vista lungo l'asse x

  - Massa = $9.10938188 \cdot 10^{-31} kg$
  - Temperatura = $46400.0K$
  - Numero di particelle reali per macro-particella = 200

- **Ioni di Argon:**
  - Carica = $-1.60217646 \cdot 10^{-19} C$
  - Massa = $1.67 \cdot 10^{-27} kg$
  - Temperatura = $300K$
  - Numero di particelle reali per macro-particella = 200

Per entrambe le specie è allocato inizialmente un numero di particelle pari a 950000, per un totale di 1.9 milioni di particelle all'inizio della simulazione.

È presente una sorgente che introduce nel dominio $8 \cdot 10^{12}$ particelle al secondo, con le stesse caratteristiche delle particelle all'istante iniziale.

La simulazione ha un passo temporale di $1 \cdot 10^{-9} s$ e procede per 130 iterazioni, riproducendo quindi un periodo di $1.3 \cdot 10^{-7} s$.

7.2 Piattaforma di test

Per eseguire le simulazioni ci si è avvalsi di un sistema desktop con le seguenti caratteristiche:

- CPU: Intel Core i7-4770 (3.4 GHz)
- RAM: 16GB DDR3 1600MHz CL9
- Scheda madre: ASUS Z87-PLUS (chipset Intel Z87)
- Sistema Operativo: GNU/Linux Fedora 18 (kernel versione 3.6.10-4)

Il sistema è dotato di una GPU NVIDIA GeForce GTX 780 con le seguenti specifiche:

- 2304 CUDA Cores
- 12 Multiprocessori SMX
- 3072MiB di memoria GDDR5 su un bus 384 bit
- Clock dei processori: 863 MHz

- Clock della memoria: 6008 MHz
- Banda di memoria: 288.4 GB/s
- TDP: 250W

7.3 Analisi dei risultati

Consideriamo come istanti di riferimento per analizzare la simulazione le iterazioni 10, 20 e 130 (l'istante finale).

In corrispondenza di queste iterazioni, il controllo di conservazione di carica implementato in F3MPIC restituisce i seguenti output testuali:

F3MPIC - Iterazione 10:

```
 Global:  sum of charges 1.634185e-11 C, integral of charge density 1.634568e-11,
error 0.023427%
Species 0:  sum of charges -1.640276e-11 C, integral of charge density -1.642406e-11,
error -0.129851%
Species 1:  sum of charges 3.274461e-11 C, integral of charge density 3.276974e-11,
error 0.076738%
```

F3MPIC - Iterazione 20:

```
Global:  sum of charges 3.082892e-11 C, integral of charge density 3.086092e-11,
error 0.103785%
Species 0:  sum of charges -4.431492e-12 C, integral of charge density -4.437198e-12,
error -0.128768%
Species 1:  sum of charges 3.526041e-11 C, integral of charge density 3.529811e-11,
error 0.106925%
```

F3MPIC - Iterazione 130:

```
Global:  sum of charges 5.309613e-11 C, integral of charge density 5.316132e-11,
error 0.122787%
Species 0:  sum of charges -3.982113e-12 C, integral of charge density -3.987196e-12,
error -0.127639%
Species 1:  sum of charges 5.707824e-11 C, integral of charge density 5.714852e-11,
error 0.123125%
```

Abilitando i moduli FGM[1] leggiamo invece:

FGM - Iterazione 10:

```
Global:  sum of charges 2.068993e-11 C, integral of charge density 2.069634e-11,
error 0.030975%
Species 0:  sum of charges -1.204318e-11 C, integral of charge density -1.203677e-11,
error -0.053214%
Species 1:  sum of charges 3.273311e-11 C, integral of charge density 3.273311e-11,
error 0.000000%
```

[1]Nel corso di questo capitolo, i risultati di simulazione verranno distinti con gli identificativi "F3MPIC"ed "FGM"per semplicità di comprensione. A scanso di equivoci, è comunque doveroso precisare che i dati "FGM"provengono da simulazioni di codice F3MPIC dove le funzionalità di interpolazione, deposizione e tracking sono sostituite dalla funzione `fgm_simstep`. Il software FGM, come sempre detto, è una libreria, non un applicativo indipendente.

FGM - Iterazione 20:

```
Global:  sum of charges 3.229465e-11 C, integral of charge density 3.235067e-11,
error 0.173441%
Species 0:  sum of charges -2.949639e-12 C, integral of charge density -2.893627e-12,
error -1.898947%
Species 1:  sum of charges 3.524429e-11 C, integral of charge density 3.524429e-11,
error 0.000000%
```

FGM - Iterazione 130:

```
Global:  sum of charges 5.311000e-11 C, integral of charge density 5.351321e-11,
error 0.759186%
Species 0:  sum of charges -3.127673e-12 C, integral of charge density -2.724469e-12,
error -12.891493%
Species 1:  sum of charges 5.623768e-11 C, integral of charge density 5.623768e-11,
error 0.000000%
```

All'istante finale, i dati sono in buon accordo, a parte un errore moderatamente elevato sulla carica degli elettroni. Si potrebbe giustificare dicendo che gli elettroni sono una specie estremamente molto più mobile ed incorrono sempre in un errore maggiore, tantopiù che il risultato sugli ioni è privo di errori e apparentemente più accurato di quello restituito dal codice originale. A prima vista, FGM è uno strumento valido e sufficientemente preciso.

Questa conclusione non è esatta ed un'ispezione delle posizioni e delle velocità previste per le particelle ce ne svela il motivo. Questi dati sono elaborati con il software Gmsh e presentati nelle Figure da 7.3 a 7.14.

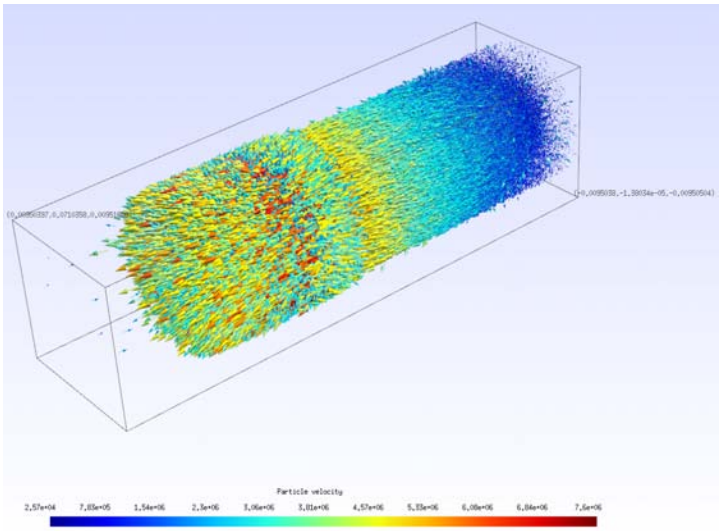


Figura 7.3: F3MPIC - Iterazione 10 - Posizioni e velocità degli elettroni

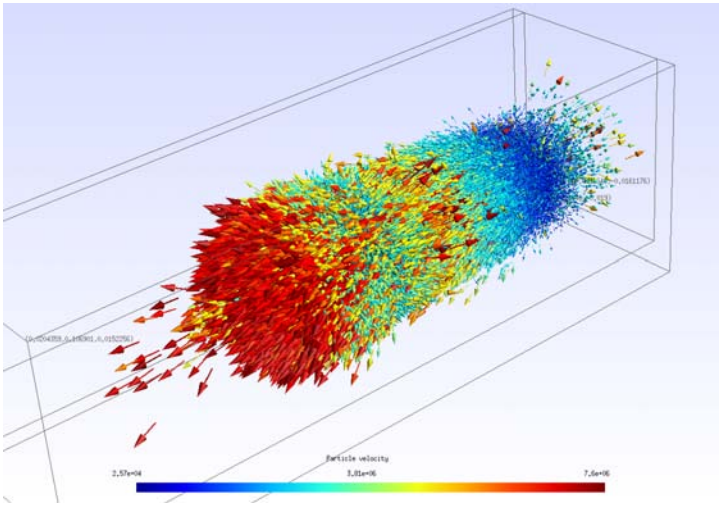


Figura 7.4: FGM - Iterazione 10 - Posizioni e velocità degli elettroni. I nodi della mesh sono visualizzati per dare un riferimento nello spazio

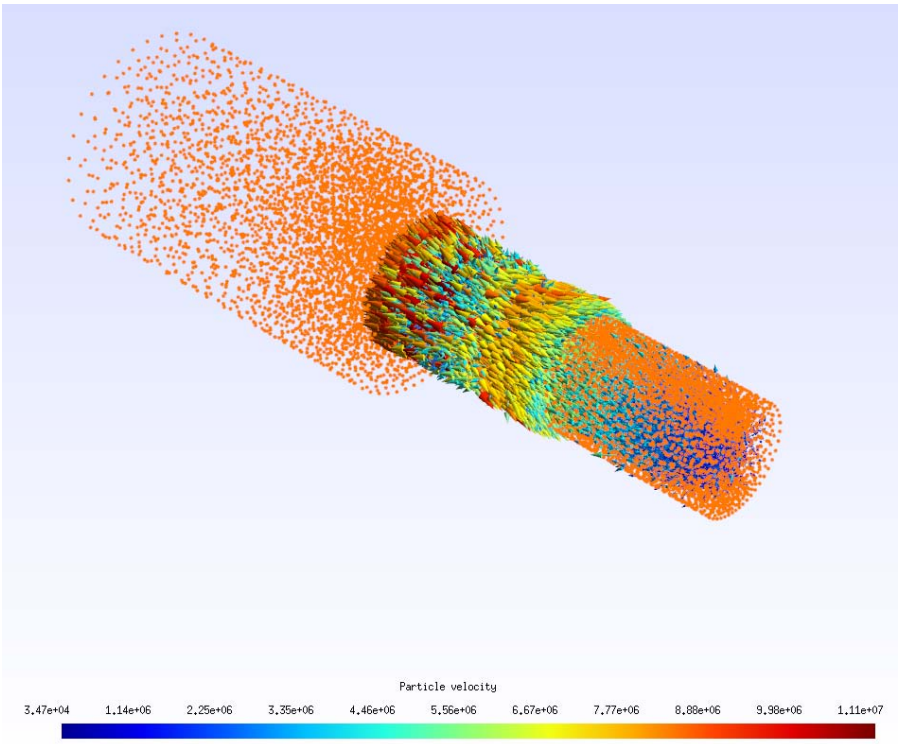


Figura 7.5: F3MPIC - Iterazione 20 - Posizioni e velocità degli elettroni. I nodi della mesh sono visualizzati per dare un riferimento nello spazio

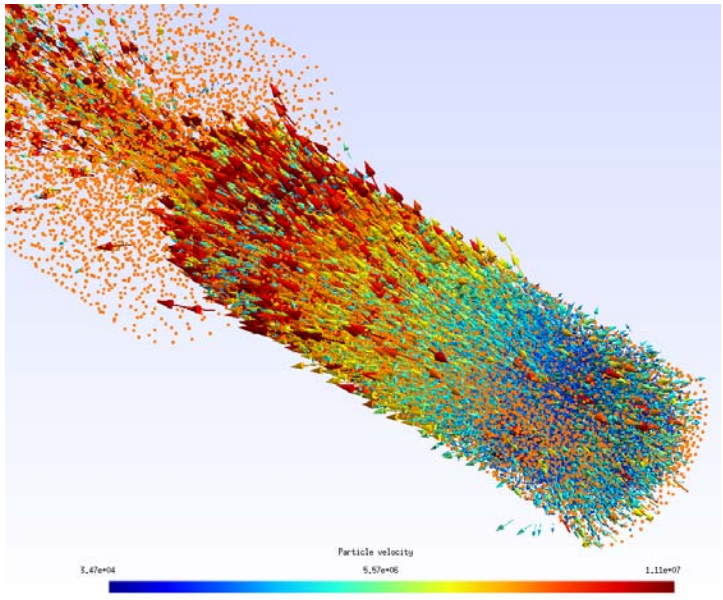


Figura 7.6: FGM - Iterazione 20 - Posizioni e velocità degli elettroni. I nodi della mesh sono visualizzati per dare un riferimento nello spazio

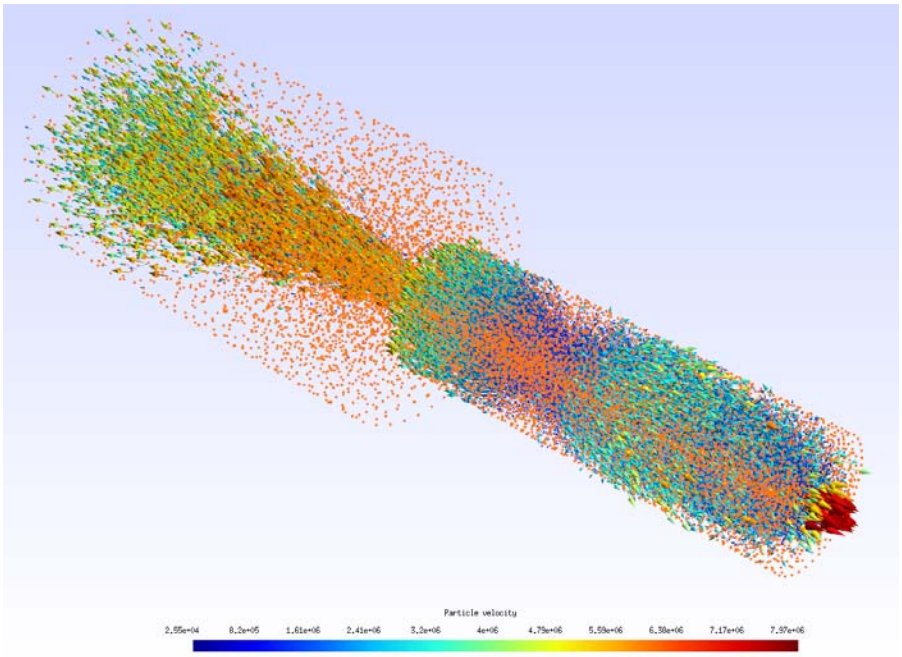


Figura 7.7: F3MPIC - Iterazione 130 - Posizioni e velocità degli elettroni. I nodi della mesh sono visualizzati per dare un riferimento nello spazio

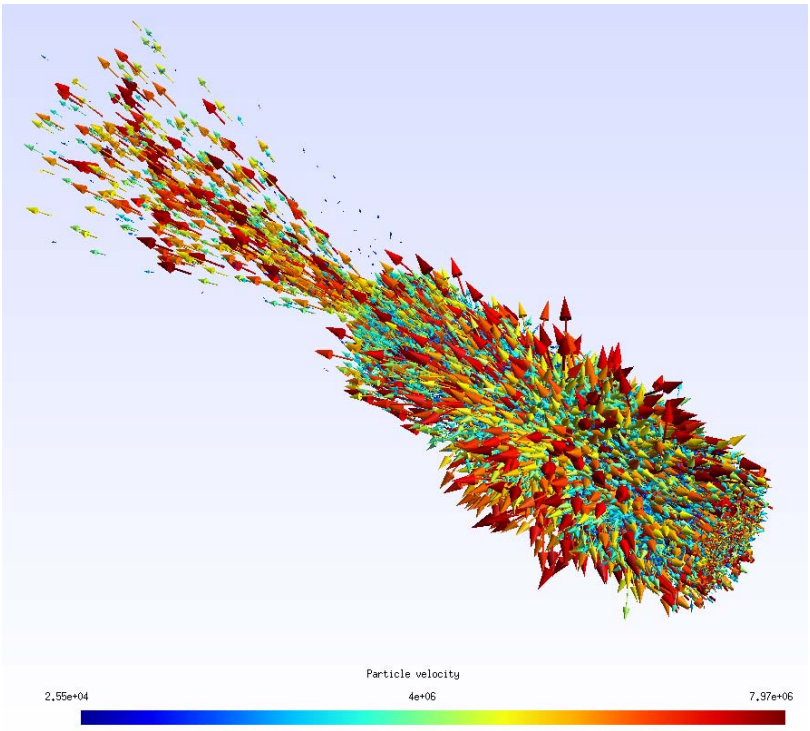


Figura 7.8: FGM - Iterazione 130 - Posizioni e velocità degli elettroni

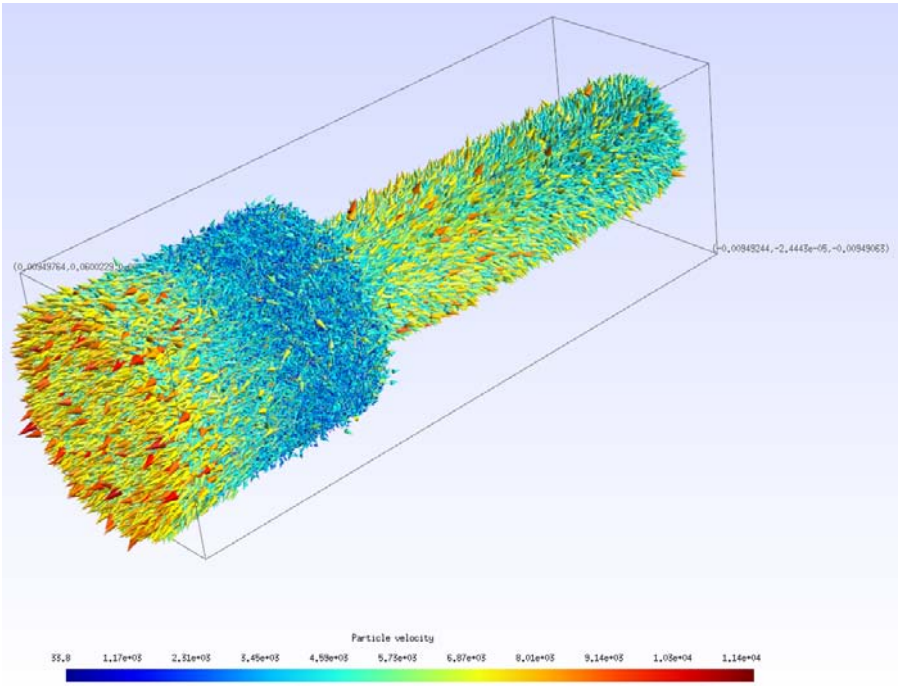


Figura 7.9: F3MPIC - Iterazione 10 - Posizioni e velocità degli ioni di Argon

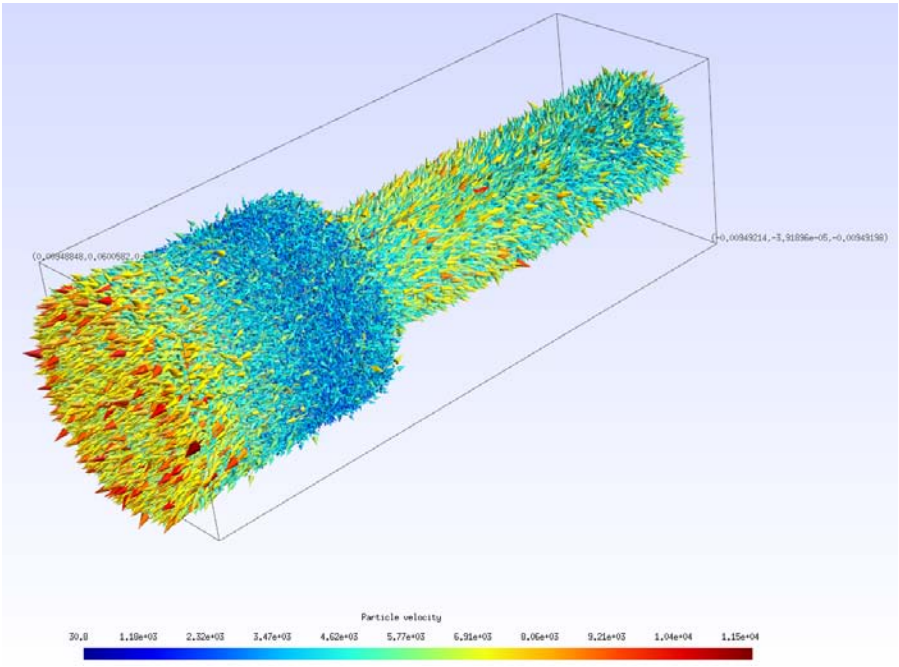


Figura 7.10: FGM - Iterazione 10 - Posizioni e velocità degli ioni di Argon

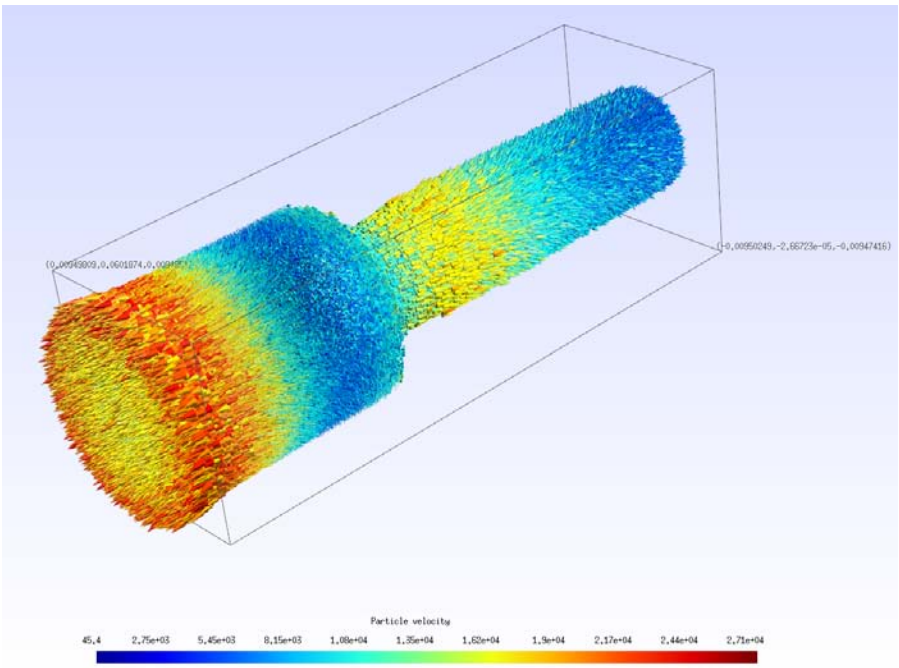


Figura 7.11: F3MPIC - Iterazione 20 - Posizioni e velocità degli ioni di Argon

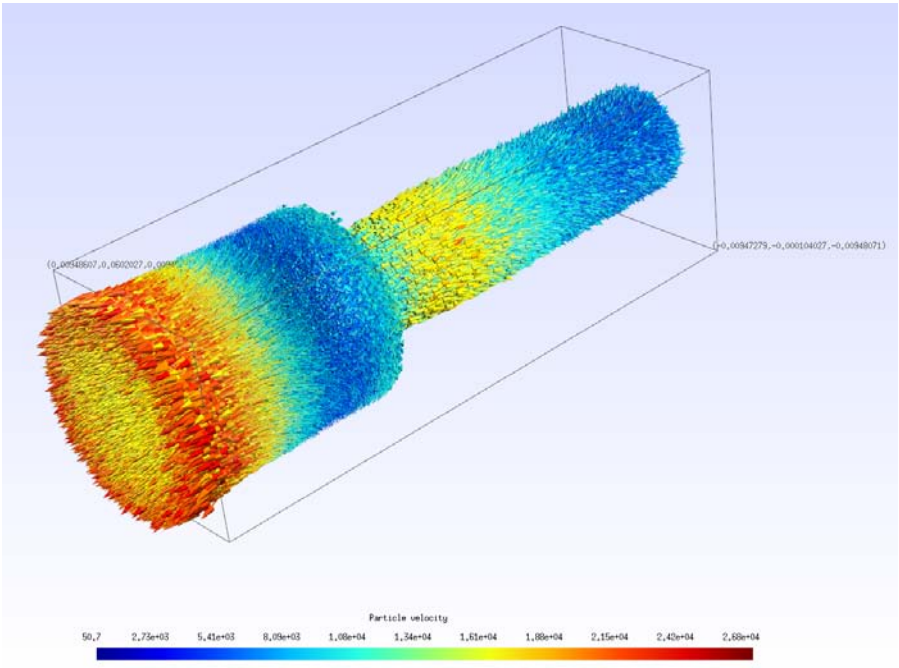


Figura 7.12: FGM - Iterazione 20 - Posizioni e velocità degli ioni di Argon

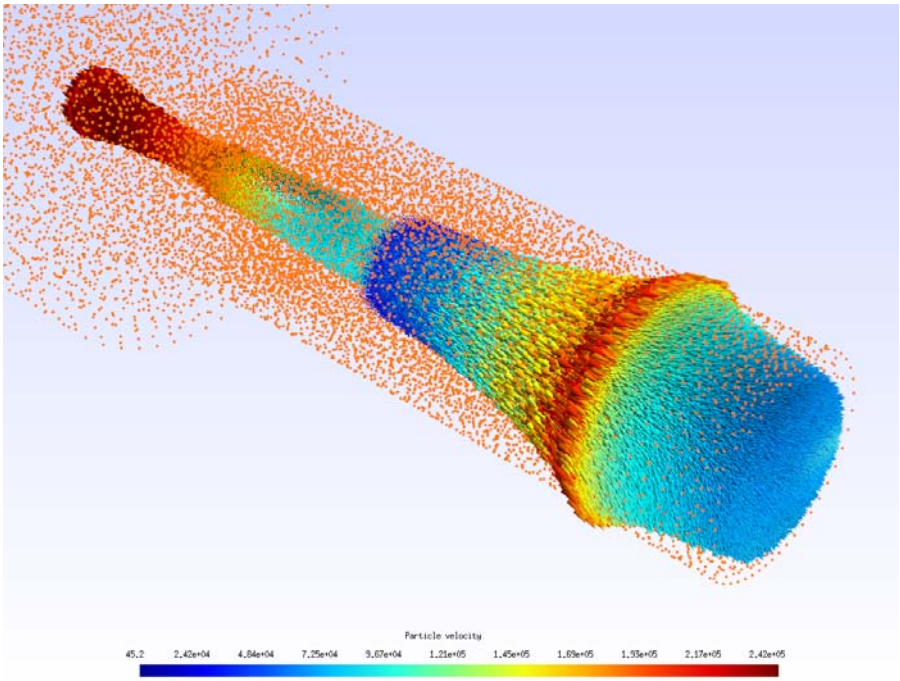


Figura 7.13: F3MPIC - Iterazione 130 - Posizioni e velocità degli ioni di Argon. I nodi della mesh sono visualizzati per dare un riferimento nello spazio

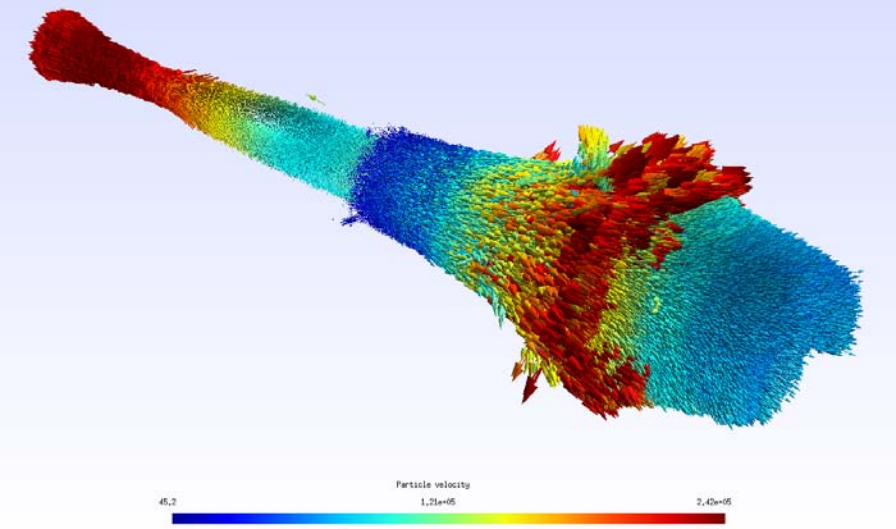


Figura 7.14: FGM - Iterazione 130 - Posizioni e velocità degli ioni di Argon

La nostra analisi ci porta a rilevare dei risultati discordanti con il riferimento offerto fa F3MPIC, nello specifico per quanto riguarda le velocità delle particelle. Questo fenomeno è molto amplificato nel caso degli elettroni, i quali sono molto più mobili degli ioni in virtù del rapporto tra la loro massa e la loro carica elettrica. Il comportamento degl iioni è invece riprodotto in maniera quasi esatta.

Il valore delle velocità è assegnato dall'integratore leapfrog-Vay, il quale però è sensibile alla fisica del problema da simulare. Poichè la validazione fisica di componenti come l'integratore e la loro attitudine a determinate classi di problemi non era considerata come uno degli obiettivi di questa esperienza, è possibile che in fase di collaudo si sia selezionato un caso con una fisica non adatta alla variante del nostro integratore. F3MPIC, invece, ha la possibilità di usufruire di numerosi codici di integrazione particellare, realizzati nel corso della sua vita operativa, e ha la possibilità di selezionare il più adatto al problema da risolvere, e di proporre risultati corretti.

Esaminiamo ora l'aspetto delle prestazioni, uno degli obiettivi primari di questo lavoro.

In Figura 7.15 è graficato il confronto fra la durata totale per iterazione della funzione `fgm_simstep()` con il tempo per iterazione necessario all'esecuzione delle analoghe funzioni F3MPIC per l'interpolazione, l'integrazione, la deposizione di carica ed il tracking (cioè quelle le cui funzionalità sono sostituite da `fgm_simstep()`). Questo paragone offre la prospettiva di un utilizzatore finale, che vede FGM come una "scatola nera", ed è agnostico riguardo al suo funzionamento.

A prima vista, non si rileva alcun miglioramento nei tempi di calcolo, anzi, il tempo medio per iterazione peggiora del 10%. Questo risultato è in disaccordo con le aspettative e le premesse teoriche del nostro lavoro.

Investighiamo più approfonditamente il funzionamento interno di `fgm_simstep()`, cronometrandone i differenti passi con riferimento all'algoritmo illustrato in Figura 6.2. I risultati sono riassunti nel grafico cumulativo di Figura 7.16.

Si osserva che in realtà l'effettiva esecuzione dei kernel di calcolo dà un contributo molto ridotto al tempo totale misurato dall'esterno di FGM. La stragrande maggioranza del tempo è occupata dai trasferimenti di memoria per scambiare le strutture relative a specie e particelle tra la GPU ed il sistema ospite; mediando i tempi su tutta la simulazione, l'88% dell'iterazione media di `fgm_simstep()` è costituito solo da copie di dati.

Se fosse possibile esprimere l'intero ciclo di simulazione in codice GPU, questi trasferimenti

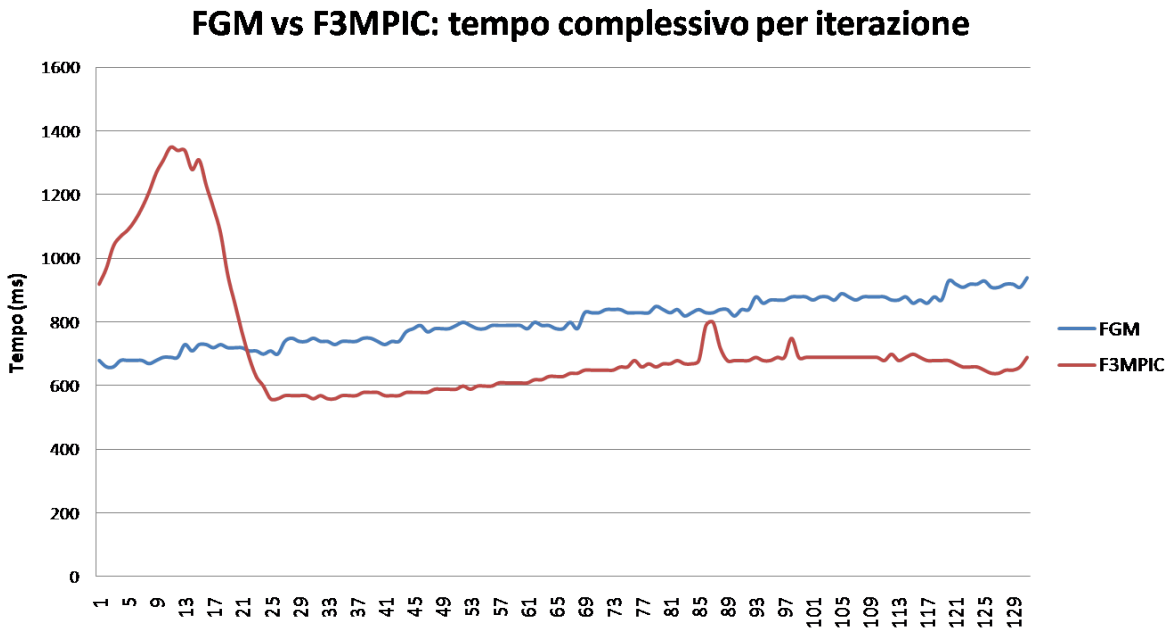


Figura 7.15: Confronto dei tempi totali per iterazione tra funzionalità analoghe di FGM e di F3MPIC

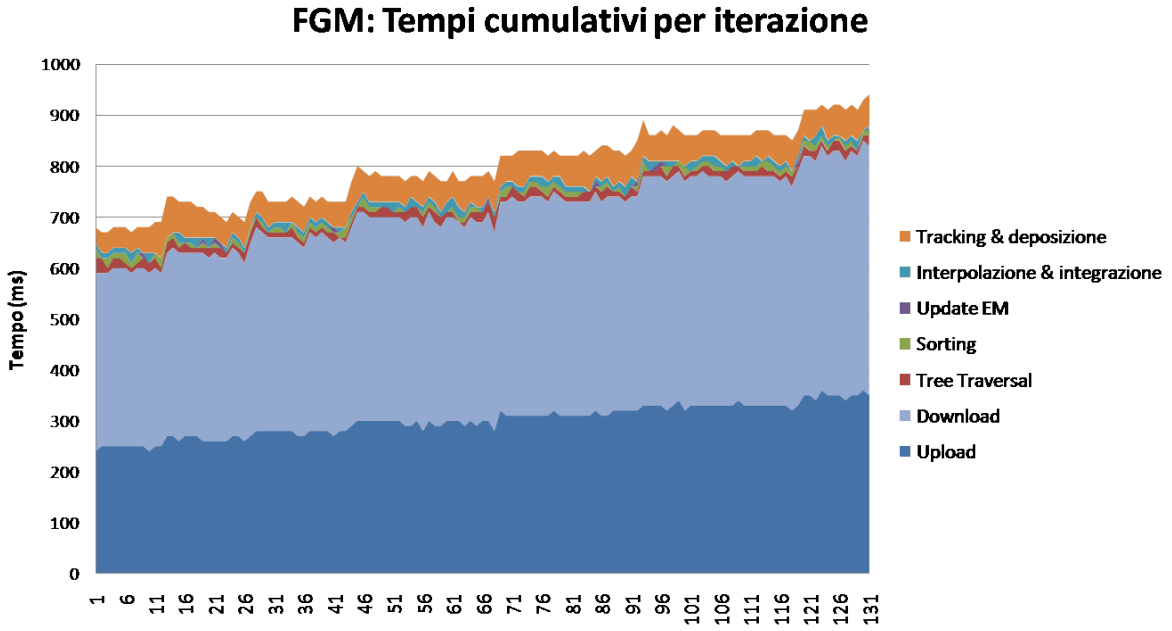


Figura 7.16: Grafico cumulativo dei tempi per iterazione delle funzioni interne a `fgm_simstep()`

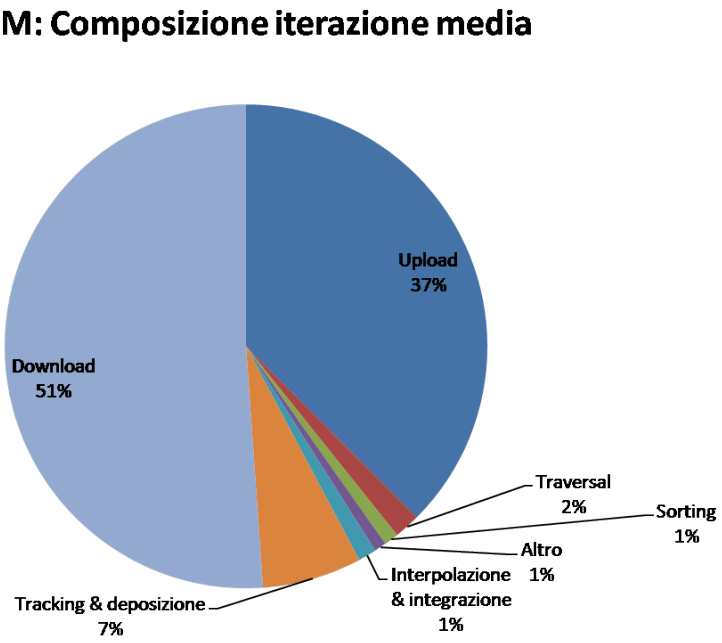


Figura 7.17: Composizione percentuale dell'iterazione media di `fgm_simstep()`

non sarebbero più necessari e si otterrebbe un'enorme miglioramento delle prestazioni. Notiamo anche che le operazioni ausiliarie introdotte per favorire il funzionamento della GPU (tree traversal, sorting) occupano un tempo quantificabile nel 3-4% dell'iterazione media. Queste funzioni, nel campo del calcolo parallelo, sono considerate delle *primitive*, ovvero costituenti elementari per costruire algoritmi più complessi: ne consegue che sono ampiamente usate e profondamente ricercate, pertanto non ci stupiamo che le loro implementazioni siano efficaci. Il dato relativo al tree traversal tuttavia non è ottimale, poichè si tratta di un implementazione di nostra realizzazione (cfr. Sezione 6.3), che non adotta nessun accorgimento per evitare la divergenza dei warp; è stata comunque preferita per la sua grande semplicità ed intuitività.

Per quantificare in termini più esatti l'accelerazione della simulazione al netto dei trasferimenti di memoria, operiamo un confronto diretto tra le funzioni che operano gli stessi stadi elementari del codice PIC: precisamente, contrapponiamo i kernel CUDA di interpolazione/integrazione e di tracking/deposizione con le corrispondenti funzionalità di F3MPIC. Le Figure 7.18 e 7.19 illustrano questi risultati.

FGM supera l'implementazione originale di F3MPIC di 11.9 volte per l'integrazione del moto delle particelle e di circa 11.36 volte per la combinazione di tracking e deposizione. L'insieme delle operazioni effettivamente utili ai fini della simulazione con le operazioni parallele ausiliarie porge un'accelerazione complessiva di 8.43 volte. È particolarmente degna di nota l'insensibilità dei tempi di esecuzione della GPU al progredire della simulazione e all'introduzione di un numero di particelle sempre maggiore nel dominio di calcolo.

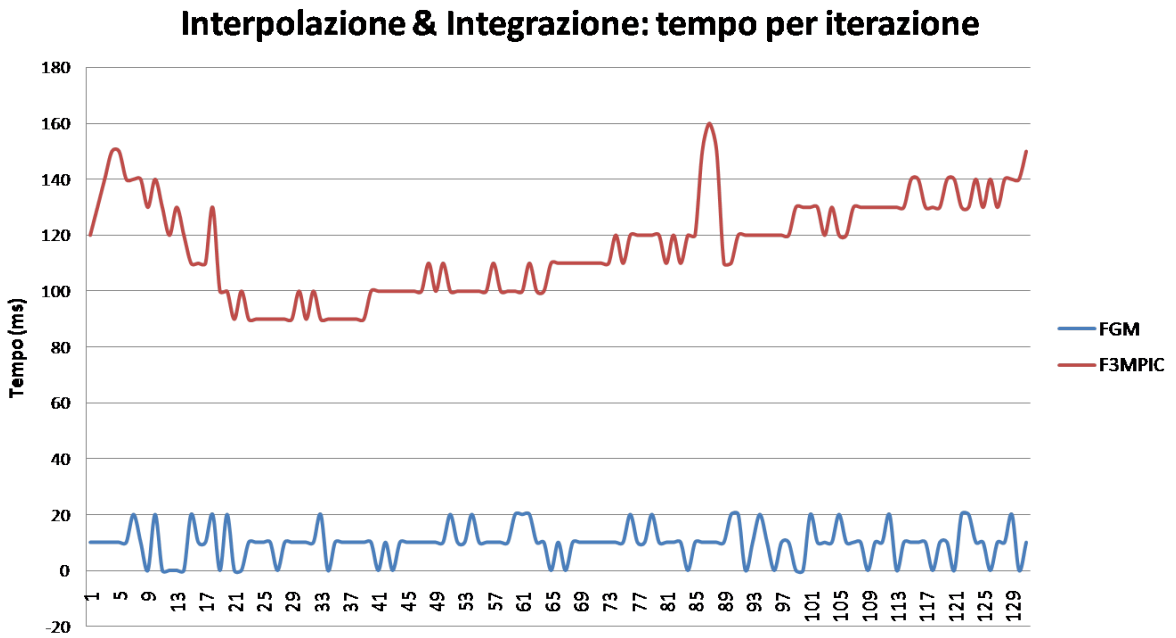


Figura 7.18: Confronto dei tempi per iterazione di interpolazione e integrazione tra FGM e F3MPIC

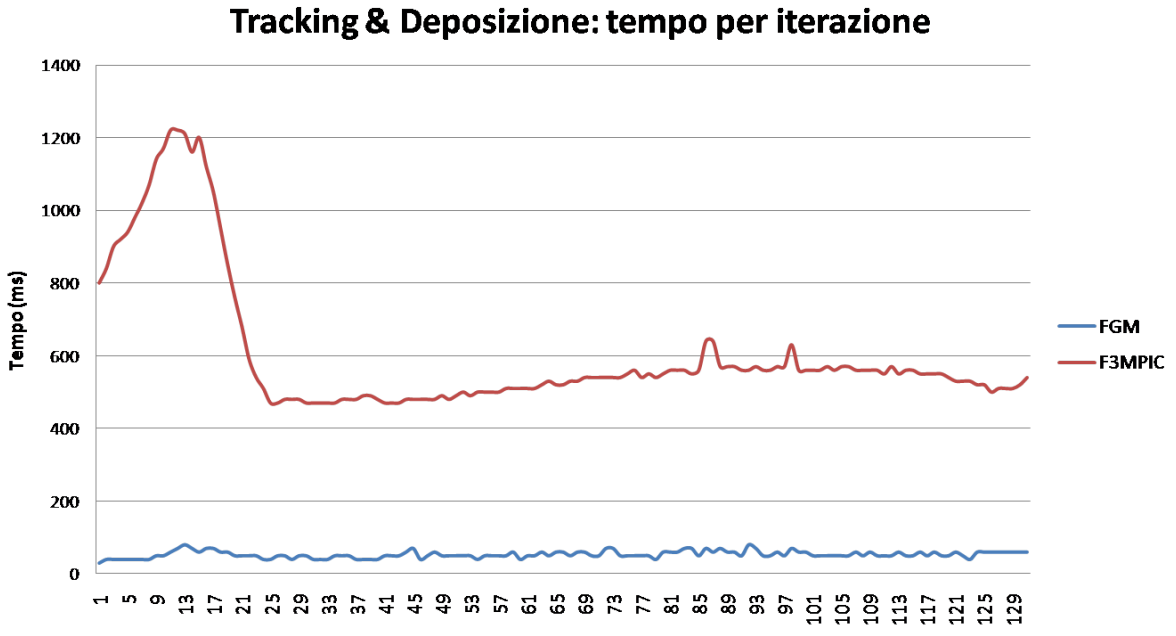


Figura 7.19: Confronto dei tempi per iterazione di tracking e deposizione tra FGM e F3MPIC

Capitolo 8

Prospettive future

Preso atto di quanto riportato nel Capitolo 7, il lavoro da svolgere nell'immediato futuro sarà quello di investigare più da vicino lo stadio di integrazione delle particelle, per ottenere dei risultati interamente in linea con l'implementazione di riferimento costituita da F3MPIC e riprodurre al meglio la fisica dei problemi richiesti.

Accanto alla correttezza dei risultati, che chiaramente nelle presenti circostanze è prioritaria, è possibile migliorare nel prossimo futuro anche le prestazioni del codice, per esempio impiegando in maniera ancora più aggressiva la tecnologia di parallelismo dinamico CUDA per con blocchi aggiuntivi di thread paralleli sostituire i cicli iterativi che elaborano sequenzialmente le particelle all'interno dei singoli nodi foglia dell'octree.

Un sensibile miglioramento è anche possibile sfruttando la memoria condivisa della GPU, come dimostrato da tutti gli autori da noi consultati. Tuttavia, nel caso del nostro algoritmo, questa funzionalità è più difficile da impiegare: il partizionamento del dominio deve essere sufficientemente raffinato in modo che i tetraedri associati a ciascun nodo foglia siano in numero tale da poter essere caricati nella shared memory senza sacrificare la occupancy del dispositivo (ovvero lanciando almeno 2 blocchi per multiprocessore). Ciò avverrebbe se un nodo foglia si intersecasse con non più di 90 tetraedri. Perchè questo avvenga, dovrebbe essere permesso all'octree di espandersi ben oltre i limiti imposti nel presente lavoro (per ottenere un esito positivo del partizionamento del dominio con massimo 5 livelli di profondità si è dovuto inserire un valore soglia di 1500 tetraedri). Aggiungendo ulteriori livelli gerarchici dell'octree, le sue dimensioni aumentano esponenzialmente, e potrebbero occupare una quantità di memoria tale da pregiudicare la possibilità di raggiungere elevati numeri di particelle nella simulazione, perchè materialmente non troverebbero spazio sulla DRAM del dispositivo. Raggiungere una soluzione di ottimo di questo problema richiede un attento studio quantitativo dei tipi di dato da utilizzare e dello spazio da essi richiesto in memoria.

È anche auspicabile una conversione su GPU dell'intera parte di simulazione legata alle particelle, lasciando alla CPU solo l'esecuzione del solutore GetDP per i campi elettromagnetici. Con questo unico accorgimento, si otterrebbe in breve tempo un miglioramento del tempo di simulazione di circa 8 volte rispetto a F3MPIC, come evidenziato nel Capitolo 7.

A lungo termine è possibile ricercare l'espansione di FGM per sfruttare GPU multiple: scegliendo opportunamente la curva a riempimento di spazio che determina la numerazione dei nodi dell'octree, sarebbe possibile distribuire in maniera soddisfacente il dominio di calcolo tra i vari dispositivi. Le zone di confine dei sottodomini sarebbero costituite da nodi foglia, che come tali avrebbero caratteristiche comuni e moli di dati simili da scambiare, favorendo l'equilibrio tra transazioni di memoria e lavoro di calcolo.

Non si devono dimenticare, inoltre, le prospettive aperte da una simulazione interamente svolta sulla GPU, che avrebbe la possibilità di integrare, per esempio, metodi per la simulazione

diretta delle collisioni tra particelle (per i quali si è già predisposti con la realizzazione dell'octree) o gli emergenti solutori basati sul metodo Discontinuous Galerkin [22], matematicamente accurati ed algoritmicamente favorevoli alla GPU.

Infine, si fa notare come il concetto generale dell'algoritmo sintetizzato ed implementato durante questa tesi non è ristretto alla simulazione del plasma, ma può essere applicato alla simulazione su mesh non strutturata di qualsiasi fenomeno fisico che possa essere modellato con uno schema ibrido Lagrangiano-Euleriano che preveda l'accoppiamento bidirezionale tra delle particelle ed una griglia spaziale. A titolo di esempio, possiamo citare fenomeni combustivi con atomizzazione dei reagenti, flussi bifase, fenomeni di erosione, e problemi di magnetofluidodinamica.

Conclusioni

In questa tesi è stato trattato lo sviluppo di moduli software che permettessero al codice di simulazione F3MPIC di avvalersi delle possibilità offerte da un coprocessore parallelo GPU.

Si è osservato che le caratteristiche fondamentali di F3MPIC (principalmente, effettuare una simulazione Particle-in-Cell non strutturata) sono in contrasto con diversi canoni di efficace utilizzo del processore grafico, rendendone inutile, se non controproducente, un'implementazione diretta.

A valle di un attento esame dello stato dell'arte della simulazione di plasma sulla GPU, si sono selezionati gli accorgimenti più validi in materia, e si sono uniti a contributi innovativi e (dove possibile) a parti del codice F3MPIC per sintetizzare un nuovo algoritmo di simulazione, che consenta di sfruttare al meglio le potenzialità della GPU.

Il principale contenuto originale è costituito dall'utilizzo innovativo di un partizionamento adattivo e gerarchico del dominio di simulazione, nella forma di un octree ottenuto con l'ausilio di una tecnologia di parallelismo dinamico. Questo octree è la spina dorsale del nuovo codice di simulazione, poichè porta un significativo numero di benefici a tutti gli stadi della simulazione, i quali a loro volta sono concepiti per servirsene. Si è anche sostituito completamente l'algoritmo di tracciamento delle particelle con una soluzione intesa per conferire maggiore consistenza ed accuratezza geometrica, unitamente ad una procedura più lineare, assecondando l'esecuzione su GPU. Un accurato tracciamento della traiettoria di ogni singola particella al fine di valutare i flussi di carica alle pareti e allo scarico è necessario per la quantificazione dell'erosione (vita media del motore) e della spinta effettiva, che rivestono un ruolo principale nella caratterizzazione delle grandezze propulsive di un sistema elettrico al plasma per applicazioni spaziali. I nuovi moduli realizzati sono stati identificati con l'acronimo FGM.

Si è rilevato uno scostamento dei risultati dai valori ottenuti con F3MPIC, usati come riferimento per la validazione. Questi scostamenti non inficiano la validità del codice, in quanto riconducibili ad un componente esterno la cui ottimizzazione non rientrava negli obiettivi che ci si era prefissi.

Dal punto di vista delle prestazioni, la mancata integrazione dell'intero ciclo di simulazione sulla GPU nega benefici in termini assoluti. Tuttavia, trascurando il tempo necessario per gli scambi dei dati con il coprocessore, l'esecuzione di funzioni fondamentali del PIC su GPU risulta complessivamente più veloce di oltre 8 volte rispetto al codice F3MPIC, pur considerando operazioni ausiliarie introdotte per rendere più congeniali i carichi di lavoro ad operazioni parallele. Questo dato assume ancora più valore se si considera che è stato ottenuto confrontando il miglior processore presente sul mercato (per questo tipo di applicazioni) con una GPU per l'intrattenimento.

Il codice inoltre si presta ad un numero significativo di futuri sviluppi: è infatti predisposto per sfruttare tecniche dalla dimostrata efficacia, ma che richiedono ulteriori accurate valutazioni per avere un'influenza positiva. In una prospettiva temporale più a lungo termine, una simulazione completamente integrata sulla GPU offrirebbe opportunità di accoppiamento con

nuovi metodi più veloci ed accurati di quelli di uso comune, sia dal punto di vista fisico che da quello matematico.

Infine, le tecniche descritte in questo lavoro possono essere usate per trattare efficacemente sulla GPU qualsiasi fenomeno fisico modellabile tramite l'interazione tra particelle lagrangiane ed una mesh (per esempio combustioni, flussi multifase, problemi di magnetofluidodinamica).

Bibliografia

[1] D.M. Goebel, I. Katz, Fundamentals of Electric Propulsion: Ion and Hall Thrusters, JPL Space Science and Technology Series, 2008.

[2] Appunti dal corso di Laboratorio di Propulsione Aerospaziale, tenuto dal prof. D. Pavarin e M. Manente presso l'Università degli Studi di Padova, A.A. 2012-2013.

[3] C. K. Birdsall, A. B. Langdon, Plasma Physics via Computer Simulation, Adam Hilger, Bristol, 1991.

[4] J.-L. Vay, Simulation of beams or plasmas crossing at relativistic velocity, Physics of Plasma **15**, 2008.

[5] G. Stantchev, W. Dorland, N. Gumerov, Fast parallel particle-to-grid interpolation for plasma PIC simulations on the GPU, Journal of Parallel and Distributed Computing, 68: 1339-1349, doi:10.16/j.jpdc.2008.05.009, 2008.

[6] X. Kong, M.C. Huang, C. Ren, V.K. Decyk, Particle-in-Cell simulations with charge-conserving current deposition on graphic procesing units, Journal of Computational Physics 230(4): 1676-1685, 2011.

[7] R. Joseph, G. Ravunnikutty, S. Ranka, E. D'Azevedo, S. Klasky, Efficient GPU Implementation for Particle-in-Cell algorithm, 2011 IEEE International Parallel & Distributed Processing Symposium, pag. 395-406, 2011.

[8] J.E. Payne, Implementation and Performance Evaluation of a GPU Particle-in-Cell Code, Tesi di Laurea, Massachusetts Institute of Technology, 2012.

[9] F. Rossi, Development of algorithms for an electromagnetic particle in cell code and implementation on a hybrid architecture (CPU+GPU), Tesi di Laurea Magistrale in Fisica, Università di Bologna, A.A. 2010-2011.

[10] H. Burau, R. Widera, W. Hönig, G. Juckeland, A. Debus, T. Kluge, U. Schramm, T.E. Cowan, R. Sauerbrey, M. Bussmann, PIConGPU - A fully relativistic particle-in-cell code for a GPU cluster, Special Issue of the IEEE Transactions on Plasma Science on Numerical Simulation of Plasma, 2010.

[11] S. Li, R. Chang, V. Lomakin, Fast Electromagnetic Integral Equation Solvers on Graphics Processing Units, in W.-M. Hwu, editor-in-chief, GPU Computing Gems: Jade Edition, capitolo 19, Morgan Kaufmann, 2012.

[12] A. Madonna, Implementazione di un integratore particellare per simulazioni di plasma su GPU con architettura NVIDIA CUDA, Relazione per il corso di Laboratorio di Propulsione Aerospaziale, Università degli Studi di Padova, A.A. 2012-2013.

[13] H. Ratschek, J. Rokne, Test for intersection between box and tetrahedron, International Journal of Computer Mathematics 65: 3-4, 191-204, 1997.

[14] N. Brünggel, Lagrangian Particle Tracking on a GPU, Tesi di Laurea, Lucerne University of Applied Sciences and Arts, 2011.

[15] G.B. Macpherson, N. Nordin, H.G. Weller, Particle tracking in unstructured, arbitrary polyhedral meshes for use in CFD and molecular dynamics, Communications in Numerical Methods in Engineering, 2008.

[16] H. M. Deitel, P. J. Deitel, C++ How to Program, 7th Edition, Prentice Hall, 2010.

[17] NVIDIA CUDA C Programming Guide Version 5.0, NVIDIA Corporation, 2012.

[18] CUDA Dynamic Parallelism Programming Guide, NVIDIA Corporation, 2012.

[19] Thrust Quick Start Guide, NVIDIA Corporation, 2012.

[20] J. Sanders, E. Kandrot, CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010.

[21] N. Whitehead, A. Fit-Florea, Precision and Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPU, NVIDIA Corporation, 2011.

[22] A. Klöckner, T. Warburton, J. Bridge, J.S. Hesthaven, Nodal Discontinuous Galerkin Methods on Graphics Processors, Journal of Computational Physics 228(2010): 7863-7882.