UNIVERSITATEA DIN BUCUREȘTI

FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ

SPECIALIZAREA INFORMATICĂ

PURE ATTENTION

Deconstructing the Magic: A White-Box Engineering Approach

Studenți

Andrei Cristian David

Sincari Sebastian George

București, 2026

Rezumat

Această lucrare adresează opacitatea framework-urilor moderne de Deep Learning ”state-of-the-art” (precum PyTorch sau TensorFlow) prin implementarea unui motor ”White-Box” în C++ și CUDA, fără librării externe. Am construit de la zero un sistem Autograd și arhitectura Transformer, integrând algoritmul Flash Attention optimizat prin Tiling în Shared Memory, acces vectorizat și Thread Coarsening. Validarea este dual: corectitudinea rețelelor Feed-Forward este verificată analitic prin metoda Celor Mai Mici Pătrate (Least Squares), iar mecanismul de Atenție este testat empiric pe task-uri de procesare de semnal (Denoising, Inversare). Rezultatul este un sistem transparent ce expune interacțiunea critică dintre algoritmii matematici și arhitectura hardware.

Abstract

This thesis addresses the opacity of modern state-of-the-art Deep Learning frameworks (such as PyTorch or TensorFlow) by implementing a ”White-Box” engine in C++ and CUDA, without external libraries. We built a custom Autograd system and Transformer architecture from scratch, integrating the Flash Attention algorithm optimized via Shared Memory Tiling, Vectorized Access, and Thread Coarsening. The validation is dual: Feed-Forward network correctness is analytically verified using the Least Squares method, while the Attention mechanism is empirically tested on signal processing tasks (Denoising, Inversion). The result is a transparent system that exposes the critical interaction between mathematical algorithms and hardware architecture.

Cuprins

Capitolul 1

Introducere

1.1 Problema Generală

Ascensiunea rapidă a domeniului Deep Learning a fost catalizată de existența unor biblioteci software extrem de performante, precum PyTorch [1] sau TensorFlow. Acestea au democratizat accesul la tehnologie, permițând cercetătorilor și inginerilor să construiască arhitecturi complexe, precum Transformer, folosind blocuri pre-definite. Totuși, acest nivel înalt de abstractizare vine cu un cost ascuns: transformarea proceselor computaționale fundamentale într-o "cutie neagră" (black box).

Motivația centrală a acestei lucrări este pur educațională: dorința de a demistifica "magia" din spatele antrenării rețelelor neurale. Nu ne propunem să construim un framework care să concureze cu soluțiile industriale în termeni de viteză, performanță sau precizie a calculelor.

1.2 Obiective

Pentru a adresa această problemă, am dezvoltat *PureAttention*, o implementare "White-box" a mecanismului de atenție și a funcționalităților necesare pentru a construi. Această lucrare își propune atingerea următoarelor obiective concrete:

1. **Implementarea End-to-End:** Dezvoltarea unui sistem complet care pornește de la ecuațiile matematice ale Transformer-ului și ajunge la execuția pe GPU, fără a depinde de biblioteci externe de Deep Learning.

2. **Flash Attention:** Implementarea și explicarea algoritmului Flash Attention V2, cu accent pe tehnicile de *Tiling* și gestiunea memoriei SRAM.

3. **Validarea Matematică:** Demonstrarea corectitudinii implementării prin compararea rezultatelor cu soluții analitice (Least Squares) și cu referința PyTorch.

4. **Analiza Performanței:** O evaluare onestă a compromisurilor dintre o implementare didactică și una industrială, cuantificând impactul optimizărilor asupra throughput-ului (GFLOPS).

Capitolul 2

Preliminarii

Fundamente Teoretice și Hardware

Backpropagation. Optimizarea rețelelor neurale se bazează pe diferențierea automată a Grafului de Computație, aplicând Regula Lanțului (Chain Rule):

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial y} \cdot \ldots \cdot \frac{\partial f_i}{\partial w_i}$$

Mecanismul de Atenție. Modelul Transformer [2] înlocuiește recurența cu Self-Attention. Pentru o secvență de lungime N și dimensiune d, proiecțiile $Q, K, V \in \mathbb{R}^{N \times d}$ sunt corelate prin:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V$$

Calculul matricei de scoruri $S = QK^T \in \mathbb{R}^{N \times N}$ impune o complexitate $O(N^2)$ în timp și memorie, devenind principalul bottleneck pentru secvențe lungi.

Arhitectura GPU. Performanța este limitată de lățimea de bandă a memoriei (Memory Wall). Deși GPU-urile moderne excelează la operații matriceale dense (GEMM) prin Tensor Cores, atenția implică operații non-GEMM (Softmax, normalizări) care sunt memory-bound. GPU-ul expune o ierarhie critică: **HBM** (capacitate mare, latență mare) vs. **SRAM** (aprox. 100KB/SM, latență minimă). Optimizarea kernel-urilor constă în minimizarea acceselor la HBM prin tiling în SRAM.

Stabilitate Numerică. Implementarea naivă a Softmax este instabilă numeric din cauza exponențialei. În practică, se utilizează "Safe Softmax", scăzând maximul pe rând (m) înainte de exponențială: $e^{x_i - m}$. Această proprietate de invarianță la translație este esențială pentru algoritmul Flash Attention.

Validare Teoretică. Pentru verificarea corectitudinii implementării mecanismului Feed Forward, comparăm rezultatele rețelei cu soluția analitică Least Squares, care minimizează eroarea pătratică medie $MSE = \frac{1}{N}\sum(y_i - \hat{y}_i)^2$ printr-o formă închisă, oferind un "Ground Truth" matematic absolut pentru problemele de regresie liniară.

Capitolul 3

Arhitectura Sistemului și Implementare

3.1 Arhitectura Multi-Layer Perceptron și Motorul de Autograd

Deși arhitectura stratului Transformer necesită strict o componentă de tip Feed-Forward, în cadrul acestui proiect am implementat funcționalitățile complete ale unui Multi-Layer Perceptron (MLP), capabil să susțină inclusiv task-uri de regresie. Această rețea procesează reprezentările rezultate din stratul de atenție, având o structură compusă din straturi liniare complet conectate, unde transformarea aplicată fiecărui strat este definită prin relația: $y = \sigma(Wx + b)$.

3.1.1 Inițializarea He și Funcția de Activare ReLU

Pentru a menține varianța semnalului constantă între straturi, am implementat inițializarea He (Kaiming), optimizată pentru funcții de activare non-liniare de tip ReLU. Formula de calcul pentru distribuția ponderilor W este:

$$W \sim \mathcal{N}\left(0, \sqrt{\frac{2}{n_{in}}}\right) \tag{3.1}$$

unde n_{in} reprezintă numărul de unități de intrare ale stratului. Această inițializare este cuplată cu funcția de activare Rectified Linear Unit (ReLU), definită matematic prin:

$$f(x) = \max(0, x) \tag{3.2}$$

3.1.2 Funcția de Cost MSE și Optimizatorul Adam

Evaluarea performanței modelului se realizează prin funcția de cost Mean Squared Error (MSE):

$$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{3.3}$$

Pentru minimizarea acestei funcții, am utilizat optimizatorul Adam (Adaptive Moment Estimation), care combină avantajele algoritmilor AdaGrad și RMSProp. Adam calculează rate de învățare adaptive pentru fiecare parametru folosind estimări ale primului moment (media) și celui de-al doilea moment (varianța necentrată) ale gradienților:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \tag{3.4}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t \tag{3.5}$$

unde g_t este gradientul la pasul curent, η este rata de învățare, iar $\hat{m}_t$ și $\hat{v}_t$ sunt momentumurile corectate pentru bias.

3.1.3 Motorul de Autograd și Graful de Computație

Elementul central al implementării este motorul de *Autograd*, care gestionează un graf aciclic orientat (DAG) al operațiilor. Fiecare tensor stochează o referință către funcția care l-a generat, permițând reconstrucția dinamică a grafului în timpul etapei de *forward*. La declanșarea etapei de *backward*, graful este parcurs în ordine topologică inversă, aplicând regula lanțului (chain rule) pentru a propaga gradienții de la funcția de cost către parametrii învățabili.

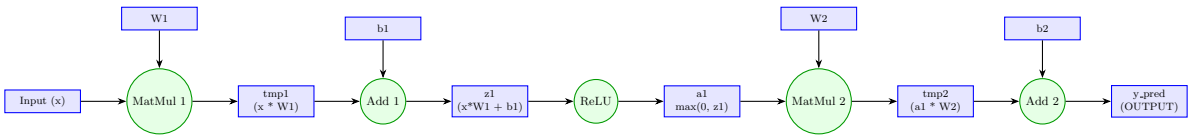

Figura 3.1: Motorul de Autograd și Graful de Computație

3.2 Implementarea Flash Attention

3.2.1 Structura Datelor și Lansarea Kernel-ului

Kernelul Flash Attention este centrul unui Multihead Attention Layer și este subiectul discuției acestui subcapitol, dar mai întâi, pentru claritate voi evidenția și flow-ul preliminar kernelului Flash Attention în figura 3.2.

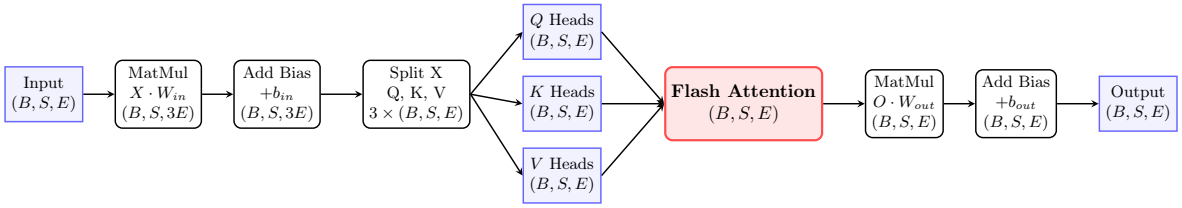

Figura 3.2: Pipeline-ul Multi-Head Attention.

Inputul este constituit din tensorii Q, K și V. Aceștia rezultă dintr-o proiecție liniară fuzionată urmată de operația de split. Pentru eficiență, proiecțiile individuale au fost unificate într-o singură operație folosind o matrice de ponderi W de dimensiune $(E, 3E)$,

rezultatul fiind ulterior structurat în formatul specific pentru intrarea în Flash Attention: (B, S, E), dar și dimensiunea de ieșire.

Structura de lansare a kernelului pe GPU este prezentată în figura 3.3, pentru mai multă claritate a structurii. Ierarhizarea a început de la vectorizarea memoriei. Cea de-a doua decizie arhitecturală din acest status de proiectare a fost Tiling Static [3], cu $B_r = 16$ și $B_c = 32$. Avem 3 motive pentru care am ales aceste valori:

- Optimizarea regiștrilor: B_c este o variabilă ce se referă la operații pe blocuri de memorie. Lucrăm cu această dimensiune pentru a optimiza și lucra eficient cu regiștrii GPU.
- Calcul static al offset-urilor: optimizare prin eliminarea operațiilor de tipul Pointer Arithmetics la Runtime.
- Template Specialization: scopul a fost pentru a lăsa compilatorul să facă Loop Unrolling complet, instrucțiunile Branch nefiind benefice întrucât golesc pipeline-ul de execuție de comenzi, până se decide condiția, deci nu se pot face optimizări hardware de tipul Instruction Level Parallelism.

Este important de menționat că modul în care se calculează dimensiunea gridului de lansare a kernelului se calculează astfel încât să nu rămână zone de memorie neacoperite.

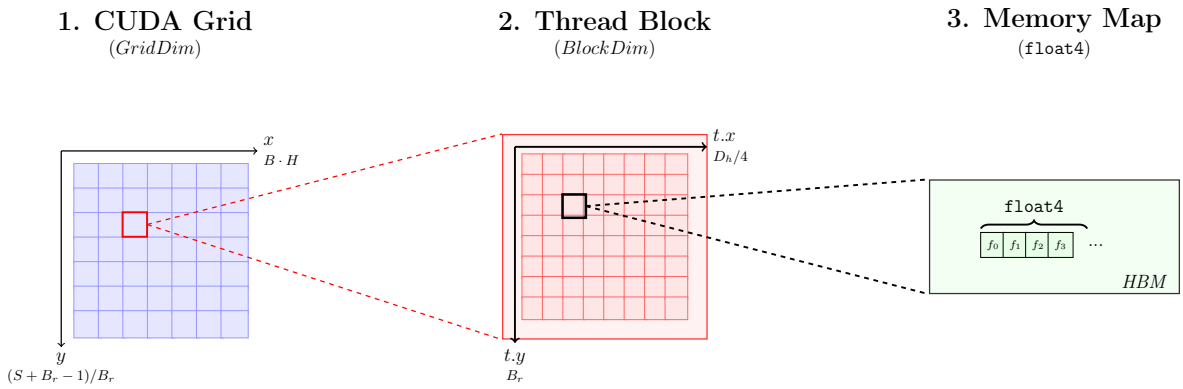

Figura 3.3: Vizualizarea Ierarhică: (1) Grid-ul lansează blocuri. (2) Un Thread Block conține o matrice de fire de execuție. (3) Fiecare fir accesează vectorizat memoria(`float4`).

Verificarea se face numărând câte float-uri sunt acoperite de thread-uri:

$$\underbrace{(B \cdot H \cdot \lceil S/B_r \rceil)}_{\text{Grid Size}} \cdot \underbrace{\left(\frac{D_h}{4} \cdot B_r\right)}_{\text{Block Size}} \cdot \underbrace{4}_{\text{Floats/Thread}} \geq B \cdot H \cdot S \cdot D_h$$

3.2.2 Flash Attention Kernel

Mecanismul de atenție a fost formalizat matematic în acest fel: $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$, dar o implementare naivă a acestei formule este costisitoare din punct de vedere al memoriei, datorită eventualei stocări intermediare a matricei S, dar și computațional, datorită Softmax-ului care în aparență este o operație secvențială. Partea revoluționară a Kernelului Flash Attention 2 [4] este mecanismul de Online Softmax [5]

care permite calcularea Softmax-ului într-o manieră paralelizată, deci o saturare mai eficientă a GPU-ului. Online Softmax este formalizat în această formă:

$$\mathbf{O}^{(2)} = \begin{bmatrix} \frac{\ell_1^{(1)}}{\ell_1^{(2)}} & & 0 \\ & \ddots & \\ 0 & & \frac{\ell_{B_r}^{(1)}}{\ell_{B_r}^{(2)}} \end{bmatrix} e^{m^{(1)}-m^{(2)}}\mathbf{O}^{(1)} + \begin{bmatrix} \frac{1}{\ell_1^{(2)}} & & 0 \\ & \ddots & \\ 0 & & \frac{1}{\ell_{B_r}^{(2)}} \end{bmatrix} e^{\mathbf{S}^{(2)}-m^{(2)}}\mathbf{V}^{(2)}$$

În această relație de recurență, $\mathbf{O}^{(2)} \in \mathbb{R}^{B_r \times d}$ reprezintă blocul de output actualizat la pasul curent, iar $\mathbf{O}^{(1)}$ este acumulatorul ce conține rezultatul parțial din iterația precedentă (calculat pe blocurile anterioare de K și V). Termenul $\mathbf{S}^{(2)}$ desemnează scorurile de atenție ($Q \cdot K^\top$) calculate pentru blocul curent. De asemenea, vectorul $\ell \in \mathbb{R}^{B_r}$ menține suma exponențiaților pentru normalizare, în timp ce $m \in \mathbb{R}^{B_r}$ reține maximul pe fiecare rând necesar pentru a asigura stabilitatea numerică.

Din punct de vedere al implementării, algoritmul adoptă o strategie *Output-Stationary*: blocurile de interogări Q sunt încărcate în regiștri și menținute constante pe durata iterației prin blocurile de chei și valori (K, V), minimizând astfel traficul cu memoria HBM. Matricea de output este scrisă în memoria globală o singură dată, la finalul procesării tuturor blocurilor, ceea ce reduce drastic lățimea de bandă consumată.

Calculul scorurilor de atenție include aplicarea factorului de scalare $1/\sqrt{d}$ pentru stabilitatea gradienților, precum și mascarea explicită a zonelor de padding (unde indexul coloanei depășește lungimea secvenței L).

Având în vedere cele mai sus menționate, fluxul de execuție al kernelului este prezentat în figura 3.4.

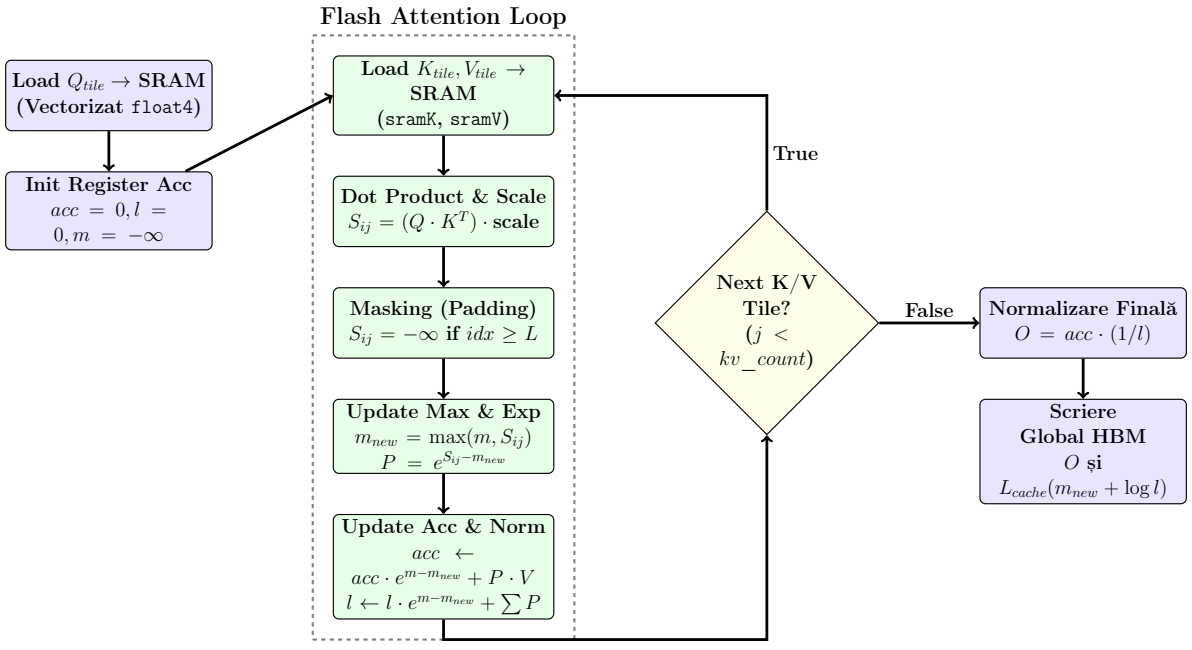

Figura 3.4: Fluxul de Execuție al Kernelului.

3.2.3 Backpropagation și Recomputare

Pentru faza de backward, am adoptat strategia de Recomputare [4]. Deoarece matricea de atenție P nu este salvată în HBM din lipsă de memorie, aceasta este recalculată on-the-fly folosind blocurile Q, K, V și factorii de normalizare L_{cache} (salvat din forward) și termenul de corecție Δ (calculat la începutul backward-ului).

Kernelul implementează inversarea buclelor față de forward: paralelizăm grid-ul pe blocurile de K și V, iterând intern prin toate blocurile de Q. Această structură este critică pentru a menține gradienții dK și dV în regiștri/SRAM.

Ecuația gradientului local este:

$$dS_{ij} = P_{ij} \cdot (dP_{ij} - \Delta_i), \quad \text{cu} \quad \Delta_i = \sum_{k=1}^{D_h} dO_{ik} \cdot O_{ik}$$

Ulterior, gradienții finali pentru intrări sunt obținuți prin proiecția erorii înapoi prin multiplicările matriciale:

$$dQ = dS \cdot K, \quad dK = dS^\top \cdot Q, \quad dV = P^\top \cdot dO$$

Fluxul de execuția a kernelului de backward este prezentat în figura 3.5.

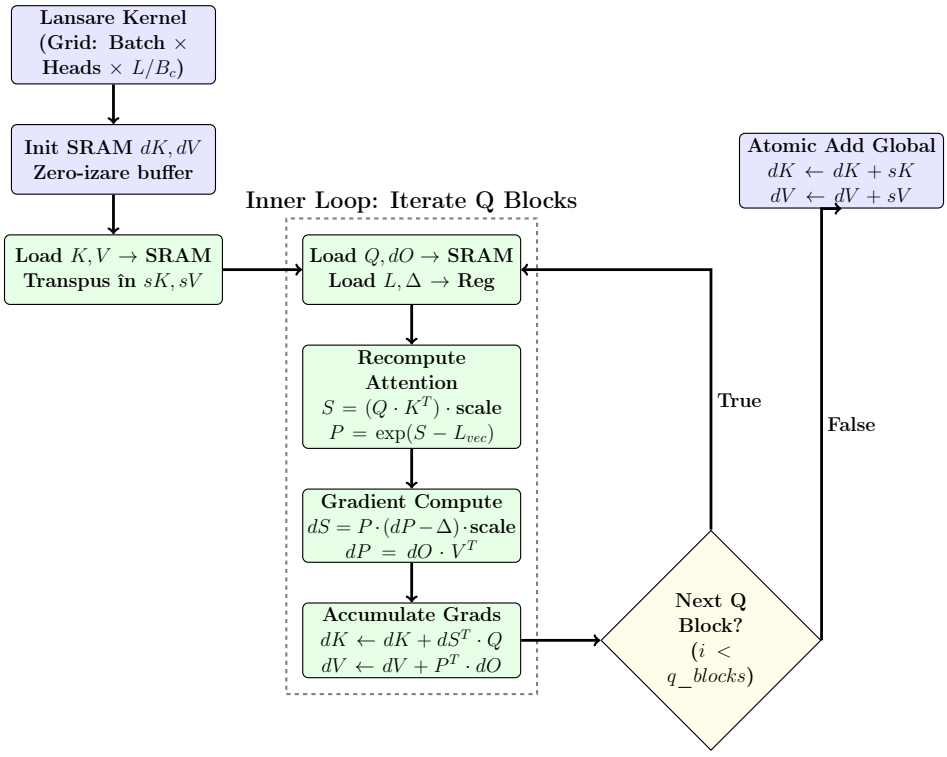

Figura 3.5: Fluxul de Execuție Backpropagation.

3.3 Arhitectura GPU și Tehnici de Optimizare

Pentru a motiva deciziile de implementare și optimizările realizate în kernel-uri, o să prezentăm pe scurt arhitectura hardware a GPU-urilor și modelul de programare CUDA [6].

3.3.1 Arhitectura Hardware

Unitatea fundamentală de calcul într-un procesor grafic NVIDIA este Streaming Multiprocessor (SM). Un GPU este compus din sute de astfel de SM-uri, fiecare conținând nuclee de calcul (CUDA cores). Gestionarea eficientă a memoriei este un factor important al performanței în programarea CUDA. Ierarhia memoriei este structurată astfel, în ordinea descrescătoare a vitezei și crescătoare a latenței:

- Regiștrii: sunt cea mai rapidă formă de memorie fiind privată fiecărui thread.
- L1 Cache (Shared Memory): este o memorie rapidă, situată fizic pe SM, partajată între toate firele de execuție dintr-un bloc și funcționează ca un cache programabil.
- L2 Cache: este o memorie partajată între toate SM-urile, neprogramabilă.
- Global Memory: Memoria principală a plăcii video, are capacitate mare, dar și o latență ridicată și o lățime de bandă limitată comparativ cu memoria on-chip.

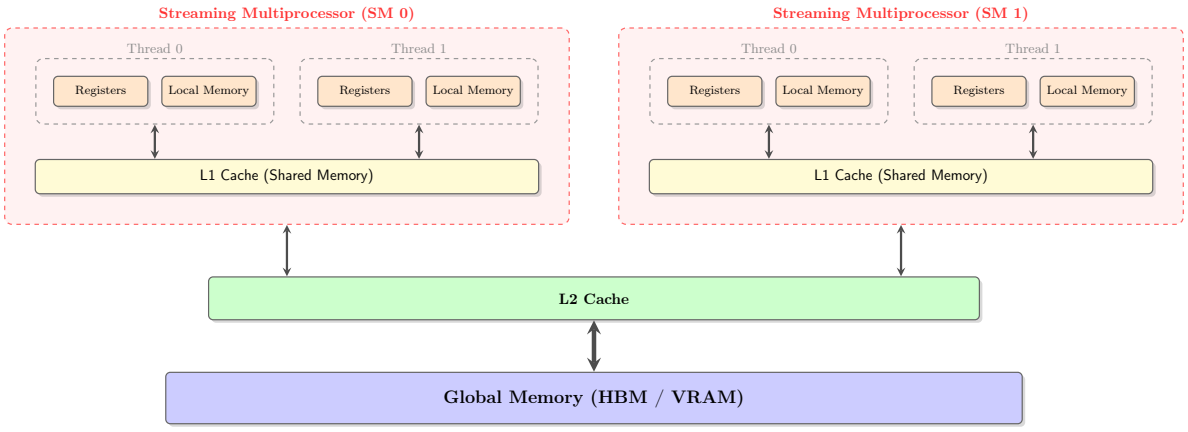

Figura 3.6: Memory hierarchy

3.3.2 Modelul de Execuție CUDA

Arhitectura CUDA expune o ierarhie de execuție care mapează resursele hardware:

- Grid: Totalitatea firelor de execuție lansate pentru un kernel. Grid-ul este divizat în blocuri (Blocks).
- Block: Un grup de fire de execuție care sunt programate pe același SM. Firele dintr-un bloc pot coopera folosind memoria partajată și bariere de sincronizare.
- Warp: Un bloc este divizat în warp-uri (de obicei în grupuri de 32). Un warp execută aceeași instrucțiune simultan (SIMT - Single Instruction, Multiple Threads).

Indexarea firelor se realizează folosind variabile native CUDA (`threadIdx`, `blockIdx`, `blockDim`, `gridDim`). De exemplu, într-o matrice, poziția globală (x, y) se poate calcula astfel:

$$x = \text{blockIdx.x} \cdot \text{blockDim.x} + \text{threadIdx.x} \tag{3.6}$$

$$y = \text{blockIdx.y} \cdot \text{blockDim.y} + \text{threadIdx.y} \tag{3.7}$$

Eficiența kernel-ului depinde de accesul memoriei, iar pentru memoria globală este crucial conceptul de *memory coalescing*. Acesta permite hardware-ului să combine cererile de memorie ale firelor de execuție, într-un număr minim de tranzacții, dacă adresele accesate sunt contigue și aliniate.

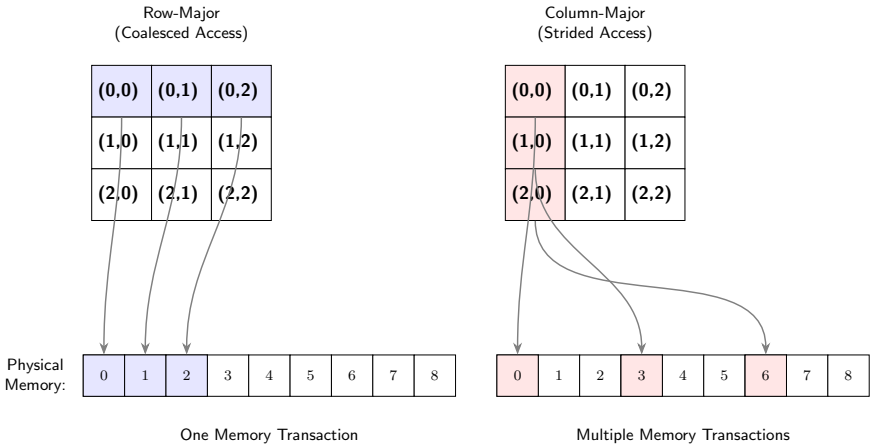

Figura 3.7: Memory Coalescing

3.3.3 Tehnici de Optimizare Implementate

Pentru implementarea de kernel-uri CUDA, am aplicat o serie de optimizări menite să maximizeze utilizarea resurselor și a lățimii de bandă.

Shared Memory Tiling

Accesarea repetată a datelor din memoria globală este costisitoare. Prin tiling, încărcăm blocuri de date din memoria globală în cea partajată. Această abordare este posibilă matematic deoarece rezultatul final se obține prin acumularea produselor parțiale calculate pe fiecare bloc. Odată ce datele au fost încărcate în *Shared Memory*, firele de execuție le pot folosi de mai multe ori cu latență minimă. Din punct de vedere hardware, memoria partajată este organizată în *Banks* (în general 32). Dacă mai multe fire dintr-un warp accesează adrese diferite care sunt mapate la același bank, apare un *Bank conflict*, iar accesările devin serializate. Am structurat tile-urile și accesările pentru a minimiza aceste conflicte.

Vectorized Memory Access

Această tehnică maximizează lățimea de bandă a memoriei prin utilizarea tipurilor de date vectorizate de tip float4, permițând fiecărui fir de execuție să acceseze 16 octeți într-o singură operațiune. Optimizarea reduce semnificativ numărul total de instrucțiuni de tip load/store emise și îmbunătățește eficiența magistralei de date prin creșterea gradului de utilizare a tranzacțiilor de memorie. Prin gruparea acceselor, se minimizează overhead-ul cauzat de latența memoriei globale și se asigură o aliniere mai bună a datelor în raport cu unitățile de execuție. Această abordare este esențială pentru atingerea performanței de vârf în aplicațiile dominate de operații de transfer de date (memory-bound).

Thread Coarsening

Thread coarsening crește granularitatea procesării per fir de execuție, permițând unui singur thread să gestioneze multiple elemente de date prin stocarea produselor parțiale în regiștri locali. Această abordare facilitează reutilizarea datelor și îmbunătățește paralelismul la nivel de instrucțiune (ILP), reducând în același timp dependența de accesările repetate ale memoriei partajate. Prin procesarea mai multor elemente de către același fir, se reduce numărul total de fire de execuție active, scăzând astfel presiunea asupra planificatorului de procese (hardware scheduler). De asemenea, tehnica permite mascarea latenței operațiilor aritmetice prin intercalarea acestora cu accesul la datele stocate în regiștri.

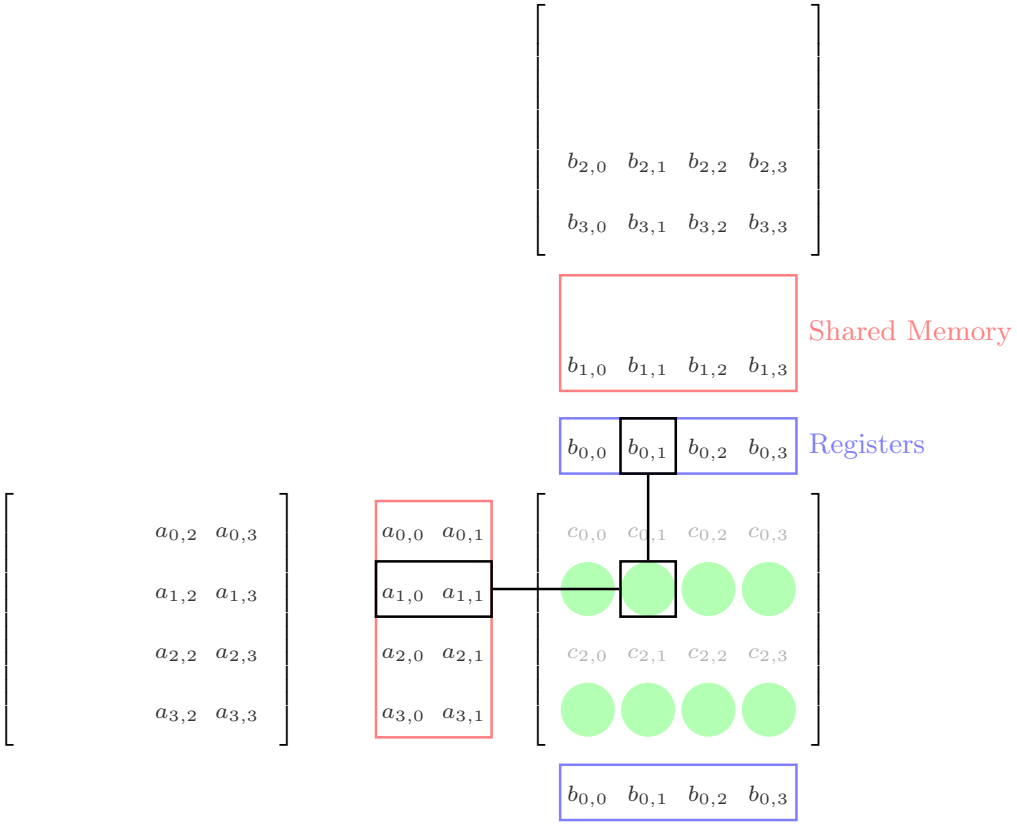

Figura 3.8: Shared Memory & Thread Coarsening

Capitolul 4

Validare Matematică și Funcțională

4.1 Corectitudinea Mecanismului Feed-Forward

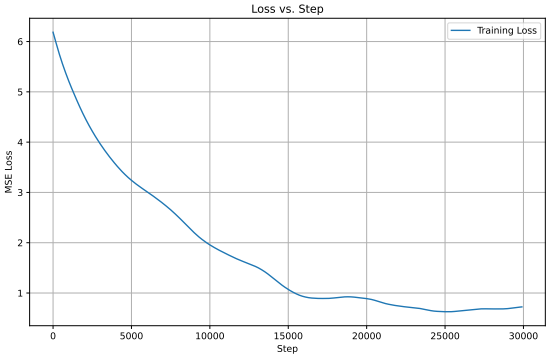

Figura 4.1: Convergență Least Squares

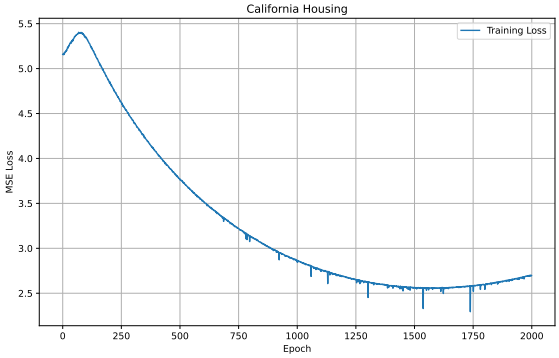

Figura 4.2: Convergență Housing

Pentru a valida corectitudinea implementării mecanismului Feed-Forward, am comparat performanța rețelei cu o soluție analitică de referință obținută prin metoda celor mai mici pătrate (Least Squares). Pentru a realiza acest experiment, am generat un singur batch de date, și rețeaua nu are nicio neliniaritate, pentru a avea sens matematic. Matricea de greutăți converge la matricea de coeficienți aferentă metodei Least Squares, iar comportamentul loss-ului este unul descrescător, cu cazuri izolate de noise sau update-uri eronate. Acest experiment este nuanțat în figura 4.1

Pe setul de date California Housing [7] (figura 4.2), observăm o convergență robustă a modelului pe date reale. Loss-ul scade consistent de la valori ridicate până la un platou în jurul valorii de 2.5, ceea ce demonstrează capacitatea rețelei de a extrage trăsături utile din date nestructurate sintetic. Spike-urile ocazionale sunt normale în lipsa unui learning rate scheduler, dar trendul general confirmă validitatea implementării pe task-uri de regresie reale. Tendința ușoară de creștere a erorii spre finalul antrenării sugerează apropierea de minimul global, dar diverge din lipsa unui Scheduler.

4.2 Convergență pe Serii de Timp

4.2.1 Convergența pe funcții Sinusoidale

În cadrul acestor experimente am generat un singur batch de date, și scopul experimentelor a fost ca transformer-layerul meu să facă overfitting pe acest batch. Rezultatele sunt prezentate în figurile 4.3 și 4.4.

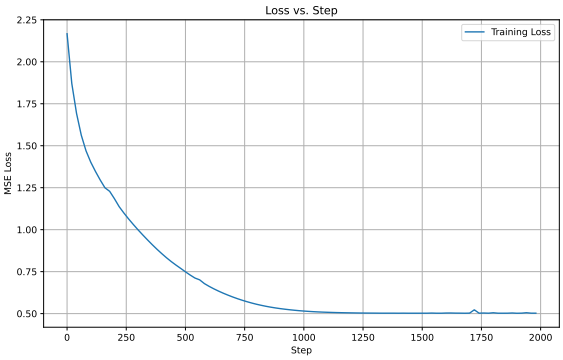

Figura 4.3: Convergență Sinus

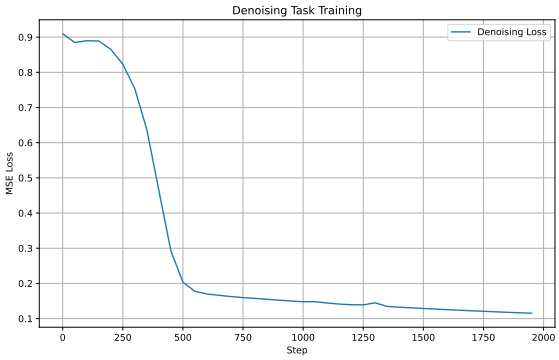

Figura 4.4: Convergență Denoising

Datele sintetice pentru cele două experimente au fost generate analitic astfel:

1. **Inversarea Sinusului**: $x_{b,s,e} = \sin(b + s + e/10)$ și $y_{b,s,e} = -x_{b,s,e}$.
2. **Denoising-ul Sinusului**: Semnalul include zgomot aditiv $\nu \sim \mathcal{U}[-0.5, 0.5]$ și fază aleatoare $\phi_b \sim \mathcal{U}[0, 2\pi]$: $x_{b,s,e} = \sin(0.1 \cdot s + 0.05 \cdot e + \phi_b) + \nu$

4.2.2 Comportamentul de convergență

Prima discuție pe care o abordez este despre Denoising-ul Sinusului, deoarece este un task destul de simplu, Transformer-ul începe cu un loss în zona de 0.9, apoi modelul caută direcția de convergență, urmată de scăderea loss-ului până la aproximativ 0.1. Acest comportament este normal și așteptat pentru un astfel de task.

În ceea ce privește Inversarea Sinusului, aici discuția nu se rezumă doar la convergență, ci și unde converge modelul. Taskul de inversare presupune ca valorile de predicție să treacă prin 0, adică media sinusoidei. În momentul în care modelul învață să prezică mereu media, adică 0, o mare parte din gradienții de pe rețea se anulează și convergența sub 0.5 este imposibilă. Mai mult de atât, $\mathbb{E}[\sin^2(x)] = 0.5$, adică acest comportament de asemenea este cel așteptat și prezis de raționamentul matematic. Modul prin care putem să remediem această situație este implementarea unui Scheduler, care ar face ca modelul să sară peste pragul de medie sau implementarea unui sistem de Gradient Clipping, care ar întârzia de asemenea update-ul gradienților pe rețea și de asemenea ar avea un efect de salt peste medie, deoarece ne-ar permite o antrenare cu un learning-rate mai mare.

Capitolul 5

Analiză Experimentală

5.1 Impactul Optimizărilor

Performanța a fost evaluată pe un GTX 1070 Ti, măsurând GFLOPS pentru kernel-uri de GEMM. Inițial, utilizarea `atomicAdd` a creat un blocaj masiv prin serializarea accesului la memorie, fiind înlocuită ulterior cu reducerea în *Shared Memory*.

Rezultatele arată că *tiling*-ul simplu depășește ușor varianta *tiled + vectorized* din cauza overhead-ului de calcul al vectorizării. Cea mai mare creștere de performanță este generată de *thread coarsening*, care maximizează reutilizarea datelor în regiștri și saturează unitățile de calcul prin reducerea presiunii asupra memoriei.

Cu toate acestea, pentru dimensiuni reduse ale matricilor, variantele cu thread coarsening au performanțe inferioare celor bazate doar pe tiling, deoarece overhead-ul devine excesiv, nucleul executând preponderent operații de transfer în regiștri în detrimentul calculelor utile.

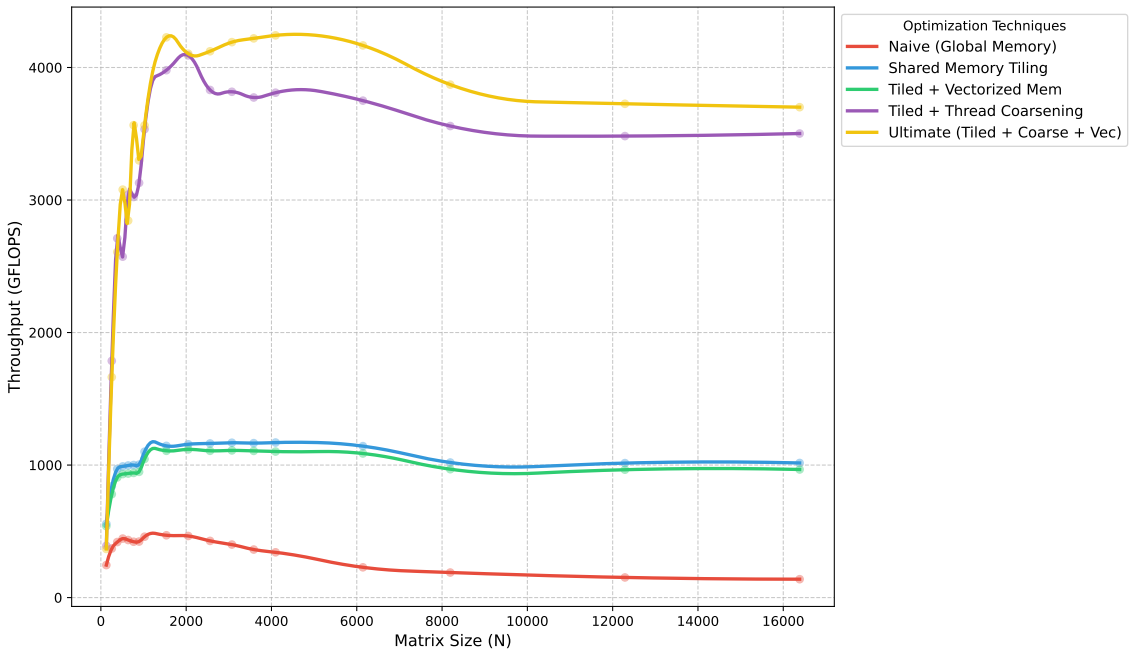

Figura 5.1: Performanța Matrix Multiplication

5.2 Profiling și Comparație cu PyTorch

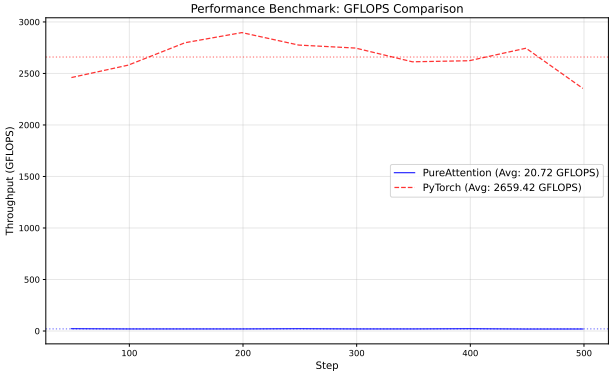

Figura 5.2: Comparație GFLOPS

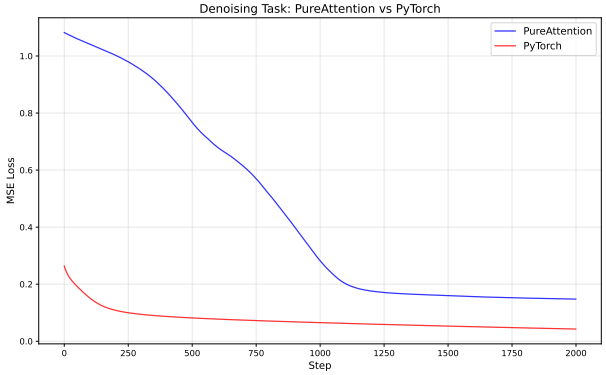

Figura 5.3: Comparație evoluție Loss

Analizând Figura 5.2, observăm o diferență semnificativă de performanță brută (throughput) între implementarea noastră (*PureAttention* - linia albastră) și PyTorch (linia roșie). Această discrepanță (medie cca. 20 GFLOPS vs 2600 GFLOPS) este justificată de doi factori arhitecturali majori.

În primul rând, PyTorch beneficiază de un *caching allocator* intern care gestionează eficient memoria VRAM, eliminând latența apelurilor sistem repetitive de tip `cudaMalloc` și `cudaFree`. În contrast, soluția noastră efectuează alocări dinamice la fiecare iterație, transformând managementul memoriei într-un bottleneck semnificativ.

În al doilea rând, framework-ul de referință utilizează kernel-uri opace (cuDNN, cuBLAS) optimizate pentru instrucțiuni specifice hardware-ului (precum *Tensor Cores*) și strategii complexe de *memory tiling*, în timp ce *PureAttention* rămâne o implementare "White-box" în CUDA C++ pur, fără dependențe externe.

Referitor la Figura 5.3, observăm o dinamică diferită de convergență între *PureAttention* și referința PyTorch, cele două modele nu ajung la aceeași valoare finală a funcției de pierdere. Acest decalaj este atribuit în principal diferențelor de stabilitate numerică: PyTorch utilizează rutine specializate pentru minimizarea erorilor de rotunjire (floating-point errors) în calculul gradienților, în timp ce implementarea noastră simplificată este mai susceptibilă la acumularea acestor erori.

Capitolul 6

Concluzii și Direcții Viitoare

Prin această lucrare am construit în scop didactic, de la zero, un motor de Deep Learning funcțional, capabil să antreneze arhitecturi Transformer moderne folosind algoritmi avansați precum Flash Attention. Prin implementarea manuală a fiecărei componente, de la graful computațional dinamic până la kernel-urile CUDA optimizate, am reușit să demistificăm o mică parte din "magia" din spatele framework-urilor moderne.

Direcții Viitoare

În urma experienței și înțelegerii căpătate pe parcursul creării acestui proiect, considerăm că următoarele îmbunătățiri ar aduce un plus de valoare și performanță:

- **Implementarea unui Caching Allocator:** Prin analiza de performanță am identificat alocarea dinamică a memoriei (apeluri repetitive de `cudaMalloc`) ca fiind un bottleneck major. O strategie de reutilizare a blocurilor de memorie ar reduce latențele semnificativ.
- **Suport pentru Tensor Cores:** Utilizarea instrucțiunilor `nvcuda::wmma` pentru a exploata unitățile matriciale specializate din arhitecturile Volta și Ampere, permițând accelerarea calculelor în precizie mixtă (FP16/BF16).
- **Kernel Fusion:** Combinarea operațiilor adiacente din graful de computație (ex: MatMul + Bias + Activation) într-un singur kernel pentru a reduce traficul de memorie globală și a crește intensitatea aritmetică.

Bibliografie

[1] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga et al., „Pytorch: An imperative style, high-performance deep learning library", în *Advances in neural information processing systems* 32 (2019).

[2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser și Illia Polosukhin, „Attention is all you need", în *Advances in neural information processing systems*, vol. 30, 2017.

[3] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra și Christopher Ré, „Flashattention: Fast and memory-efficient exact attention with io-awareness", în *Advances in Neural Information Processing Systems* 35 (2022), pp. 16344–16359.

[4] Tri Dao, „FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning", în *The Twelfth International Conference on Learning Representations*, 2024.

[5] Maxim Milakov și Natalia Gymnopoulos, „Online normalizer calculation for softmax", în *arXiv preprint arXiv:1805.02867* (2018).

[6] NVIDIA Corporation, *CUDA C++ Programming Guide*, 2023.

[7] R Kelley Pace și Ronald Barry, „Sparse spatial autoregressions", în *Statistics & Probability Letters* 33.3 (1997), pp. 291–297.