КІБЕРНЕТИКА
та КОМП'ЮТЕРНІ
ТЕХНОЛОГІЇ

УДК 519.6:004.8 DOI:10.34229/2707-451X.22.2.6

Р.О. ВДОВИЧЕНКО, В.Г. ТУЛЬЧИНСЬКИЙ

ПАРАЛЕЛЬНА РЕАЛІЗАЦІЯ РОЗРІДЖЕНО-РОЗПОДІЛЕНОЇ ПАМ'ЯТІ ДЛЯ ЗБЕРЕЖЕННЯ СЕМАНТИКИ

Запропоновано ефективну паралельну реалізацію для гібридного інтегрованого семантичного сховища CS-SDM для графічних прискорювачів на платформі NVIDIA CUDA. Проаналізовано обчислювальну складність операцій для паралельної реалізації. Запропоновано оптимізацію відновлення векторів для експериментів із послідовними пакетами даних.

Ключові слова: *GPU, CUDA, нейронна мережа, розріджено-розподілена пам'ять, асоціативна пам'ять, Compressive Sensing.*

© Р.О. Вдовиченко, В.Г. Тульчинський, 2022

Вступ. Сучасний стрімкий розвиток у застосуванні систем штучного інтелекту викликаний можливістю їхньої високоефективної програмної реалізації з використанням графічних процесорів (Graphics Processing Unit, GPU) завдяки локальності операцій з даними і однорідності алгоритмів (fine-grained parallelism).

З такої точки зору перспективною є модель розріджено-розподіленої пам'яті (Sparse Distributed Memory, SDM), запропонована П. Канервою у 1988 р. [1]. Ця модель поєднує багато властивостей природної пам'яті: узагальнюючу здатність, асоціативність, запам'ятовування послідовностей тощо. Через декілька років Льюїс Джекел опублікував альтернативну гіперплощинну конструкцію SDM [2]. За рахунок зменшення кількості порівнянь при обчисленні активації, модель Джекела споживає менше пам'яті, виконує операції запису та зчитування швидше. Пізніше було показано, що SDM Джекела узгоджена із функціонуванням мозочку ссавців, відповідно до моделі Марра – Альбуса [3]. Водночас, модель Канерви відповідає моделі імунної пам'яті Сміта – Форрест – Перельсона [4].

Бінарні розріджено-розподілені подання (Binary Sparse Distributed Representations, BSDR) [5, 6] – це феноменологічна модель різних аспектів біологічної пам'яті, підмножина векторно-символьних архітектур (Vector Symbolic Architectures, VSA [7, 8]). BSDR являє собою сімейство методів для кодування даних, наділених структурою (холістичних, ієрархічних тощо), у бінарні вектори. Важливим елементом BSDR є очищуюча пам'ять, яка відновлює кодвектори з неповних (розірджених) зразків отриманих з операції декодування (unbinding).

Ідея об'єднати SDM та BSDR виникла давно, втім, класичні конструкції SDM – не ефективні як очищуюча пам'ять для BSDR: фізична пам'ять використовується неефективно, значна частина адресного простору не використовується, операція зчитування – складна та нестійка навіть якщо розподіл відомий a priori.

Ми можемо заповнити розрив між SDM та BSDR за допомогою третьої теорії: стискуючих вимірювань (Compressed Sensing, CS). Стискаючі вимірювання запропоновані у 2004–2006 рр. Еммануелем Кандесом, Терренсом Тао та їх колегами [9, 10]. Ця теорія пропонує новий погляд на бажані умови відновлення сигналу за непрямими вимірюваннями, де замість обмеження частот використовується розрідженість сигналу. В основі відновлення сигналу методом CS лежать розв'язання недовизначеної системи лінійних рівнянь відносно норми L_0: $\|\vec{x}\|_{L_0} = |\{\vec{x_i}|\vec{x_i} \neq 0\}|$.

Результат поєднання трьох теорій – конструкція моделі CS-SDM [11, 12]. Пам'ять CS-SDM складається з трьох блоків: кодера, власне блоку пам'яті (SDM) та декодера. Кодер перетворює вхідні дані у зручну форму для зберігання у блоці пам'яті, декодер перетворює зчитані дані знову у розріджену форму. Новизна такого підходу полягає в інтеграції блоку пам'яті на основі SDM та декодера на основі CS.

У кодері M-розрядні бінарні розріджені вектори перетворюються на m-розрядні цілочисельні щільні вектори. Словник $\mathbf{\Lambda} = \{\pm 1\}^{m \times M}$ заповнюється рівномірними однаково розподіленими псевдовипадковими величинами і зберігається у звичайній пам'яті. Ця матриця відповідає умовам нормування зразків і водночас є зручною для отримання цілочисельних результатів невеликої амплітуди. Нормування відповідно переноситься на етап зчитування. Адреси у нашій конструкції залишаються розрідженими.

Як блок пам'яті використано SDM конструкції Джекела з невеликими змінами:

• оскільки вхідні дані є вже не бінарними, а цілочисельними векторами, то правило зміни лічильників у активованих комірках спрощується до простого підсумовування:

$$u_i^n(t+1) = u_i^n(t) + v_i;$$

• для використання у разі зчитування в кожній комірці передбачено додатковий 0-й розряд, до якого під час кожного запису додається 1:

$$u_0^n(t+1) = u_0^n(t) + 1;$$

• вихідні дані – теж уже не бінарні, а дійсні числа, тобто правило зчитування змінюється на:

$$v_i = \sum_{n \in A(\boldsymbol{a})} \frac{u_i^n}{u_0^n}, \; i = \overline{1, m}.$$

Останнім блоком є декодер, куди надсилається результат, зчитаний із блоку пам'яті. Декодер шукає s-розріджений розв'язок недовизначеної системи лінійних рівнянь. В експериментах ми використовували два різних методи: «жадібний» алгоритм із сімейства Matching Pursuit – CoSaMP [13] та лінійне програмування [14].

Основні операції із CS-SDM, такі як активація комірок, читання та запис, є зручними для паралельних обчислень. З цієї причини природньо застосувати графічні прискорювачі (GPU) саме для таких операцій. До безумовних переваг GPU належать:

- низька вартість;
- висока продуктивність;
- значний паралелізм.

Як обчислювальну платформу було обрано CUDA (Compute Unified Device Architecture). CUDA – це програмна модель для розробки паралельних застосунків, що розробляється компанією NVIDIA. Вона представляє собою програмну абстракцію, що дає прямий доступ до набору інструкцій GPU і паралельних обчислювальних елементів. Перевагами CUDA є:

- уніфікована пам'ять (починаючи із версії 6.0);
- швидка розділена пам'ять для взаємодії між потоками (shared memory);
- швидкі трансфери даних на GPU і назад на основний прилад.

Мова програмування для CUDA – “CUDA C/C++”, розширення C/C++. До цього розширення входять:

- специфікатори функцій, які вказують, де функція буде виконуватись і звідки може бути викликана;
- специфікатори змінних, які задають тип пам’яті, що будуть використовуватись для змінних;
- директива, що служить для запуску ядра;
- вбудовані змінні, що мають інформацію про поточний потік (зокрема, його координати в решітці та блоці);
- вбудовані макроси, що мають інформацію про GPU;
- runtime зі спеціальними типами даних та обробкою помилок.

Базовими операціями CS-SDM є:

1) ініціалізація (генерація масок, виділення пам’яті для комірок);
2) запис заданого вектора із заданою адресою до пам’яті;
3) зчитування вектора за заданою адресою із пам’яті;
4) фіналізація (звільнення пам’яті, виділеної для масок, комірок та ін.).

Можна зауважити, що і операція запису, і операція зчитування виконують пошук активованих комірок за заданою адресою. Цей пошук природно виділити в окрему операцію.

Складність паралельної конструкції алгоритмів CS-SDM будемо розглядати з точки зору граничної ефективності паралелізму (тобто припускаємо, що можемо мати будь-яку кількість паралельних обчислювальних елементів).

Алгоритм 1 (пошук активованих комірок)

Параметри:

1) адреса $\overrightarrow{address} \in \{0,1\}^L$;
2) масив із індексами активованих комірок $\overrightarrow{activated_indices} \in \mathbb{Z}_+^N$;
3) лічильник кількості активованих комірок $counter \in \mathbb{Z}_+$.

Крок 1. Паралельно по комірках перевіряємо узгодженість масок із адресою. Якщо маска узгоджена із адресою – атомарно інкрементуємо значення лічильника counter. Атомарний інкремент повертає попереднє значення лічильника, тому використовуємо його як індекс масива для запису номера поточної комірки. (Слід зауважити, що атомарні операції є блокуючими і тому часте їх використання сповільнює виконання програми. Втім, в наших експериментах середня кількість активованих комірок не була великою).

Крок 2. Маємо масив із номерами активованих комірок і їх кількість. «Хвіст» масиву залишається нульовим.

Складність (послідовно): $O(K \cdot N)$, де N – кількість фізичних комірок, K – довжина масок.

Складність (паралельно): $O(K)$, де K – довжина масок.

Зауваження. При значній кількості операцій зчитування і запису з точки зору швидкості обчислень вигідніше не виділяти і звільняти кожного разу пам’ять для цього масиву (його довжина збігається із кількістю фізичних комірок), а виділити пам’ять один раз і «обнуляти» його після кожної послідовної операції пошуку, що здійснюються при записі та читанні ($cudaMemset()$).

Алгоритм 2 (ініціалізація CS-SDM, або конструктор)

Параметри:

1) довжина масок $K \in \mathbb{N}$;
2) розрядність адрес $L \in \mathbb{N}$;
3) розрядність значень $M \in \mathbb{N}$;

4) кількість комірок $N \in \mathbb{N}$;
5) кількість CUDA-блоків $b \in \mathbb{N}$;
6) кількість CUDA-потоків на один блок $t \in \mathbb{N}, t \leq 1024$;
7) тип вектора фізичних комірок $cell_type \in \{int8, int16, int32, int64\}$;
8) тип вектора масок $index_type \in \{int8, int16, int32, int64\}$.

Крок 1. Виділяємо на GPU пам'ять для масиву фізичних комірок cells (довжина $N \cdot (M + 1)$, тип $cell_type$).

Крок 2. Виділяємо на GPU пам'ять для масиву масок indices (довжина $N \cdot K$, тип $index_type$).

Крок 3. Виділяємо на GPU пам'ять для масиву індексів активованих комірок $activated_indices$ (див. зауваження до алгоритму 1).

Крок 4. Виділяємо на GPU пам'ять для матриці стискання $\mathbf{\Lambda} \in \{\pm 1\}^{m \times L}$, заповнюємо її випадковим чином.

Крок 5. Зчитуємо маски для фізичних комірок із диску. Копіюємо їх на GPU в $indices$.

Складність: $O(N)$, де N – кількість фізичних комірок.

<u>Алгоритм 3 (запис до CS-SDM)</u>

Параметри:

1) адреса $\overrightarrow{address} \in \{0, 1\}^L$;
2) значення $\overrightarrow{value} \in \{0, 1\}^L$;
3) вага $w \in \mathbb{N}$ (за замовчуванням рівна 1).

Крок 1. Кодуємо розріджений вектор у щільний: $\vec{v} = \mathbf{\Lambda} \cdot \overrightarrow{value}, \vec{v} \in \{0, 1\}^m$.

Крок 2. Шукаємо активовані комірки (див. алгоритм 1). Якщо комірок немає – повертаємо нуль.

Крок 3. Паралельно по номерах активованих комірок додаємо до кожного розряду j активованої комірки значення $w \cdot \vec{v}_j$. До розряду $M + 1$ додаємо вагу w.

Крок 4. Повертаємо кількість активованих комірок $activations \in \mathbb{Z}_+$.

Складність (послідовно): $O(K \cdot N + M \cdot (L + N_{act})$, де K – довжина маски, N – кількість фізичних комірок, M – розрядність CS-SDM, L – довжина адреси, N_{act} – кількість активованих комірок, $0 \leq N_{act} \leq N$.

Складність (паралельно): $O(K + M)$, де K – довжина маски, M – розрядність CS-SDM.

<u>Алгоритм 4 (зчитування із CS-SDM)</u>

Параметри:

1) адреса $\overrightarrow{address} \in \{0, 1\}^L$.

Крок 1. Шукаємо активовані комірки (див. алгоритм 1). Якщо комірок немає – повертаємо нульовий вектор.

Крок 2. Виділяємо на GPU пам'ять для вектора сум $cuda_sum$ (довжина – M, тип – $float64$).

Крок 3. *Паралельно* по номерах активованих комірок додаємо до кожного розряду j вектора сум $cuda_sum$ значення $\frac{cell_j}{count}$, де $count = cell_{M+1}$ – загальна кількість векторів, записаних в цю комірку.

Крок 4. Нормуємо вектор сум $cuda_sum$, кожен розряд розділяємо на кількість активованих комірок.

Крок 5. Копіюємо $cuda_sum$ із пам'яті GPU.

Крок 6. Декодуємо $cuda_sum$ у розріджений вектор, шукаючи розв'язок недовизначеної системи лінійних рівнянь (CoSaMP або LinProg).

Складність: визначається складністю обраного алгоритму відновлення розрідженого вектора на кроці 6.

Алгоритм 5 (фіналізація CS-SDM, або деструктор)

Крок 1. Звільняємо на GPU пам’ять, виділену для масиву фізичних комірок $cells$.

Крок 2. Звільняємо на GPU пам’ять, виділену для масиву масок $indices$.

Крок 3. Звільняємо на GPU пам’ять, виділену для масиву індексів активованих комірок $activated_indices$ (див. зауваження до алгоритму 1).

Складність: $O(1)$.

Розріджені бінарні вектори було згенеровано випадковим чином, як і маски для фізичних комірок пам’яті; ці дані було збережено як бінарні файли для зручності використання. Запис і читання виконувались інкрементальними пакетами, тобто генеральна множина тестових векторів $A = \{\vec{a}_i | 1 \le i \le I, \ \vec{a}_i \in \{0,1\}^L\}$ розбивалась на J рівнопотужних множин $A_j = \left\{\vec{a}_i \mid j \cdot \frac{I}{J} < i \le (j+1) \cdot \frac{I}{J}\right\}, 0 \le j < J$. Після запису пакета за номером j із пам’яті читаємо вектори із усіх вже записаних пакетів.

Відновлення розріджених векторів проводилось кількома методами: CoSaMP та LinProg. В обох випадках використовуються бібліотеки на мові програмування Python, із CPU як обчислювальний пристрій. Для розділення обчислень на різних пристроях, читання/запис CS-SDM та подальше відновлення розріджених векторів зроблено двома окремими програмами, із використанням збереження щільних сигналів, зчитаних із CS-SDM, на диску. Така структура дозволяє експериментувати паралельно на двох різних пристроях (CPU і GPU).

Для CoSaMP використовувалась процедура, доступна у бібліотеці [15] як Python-ноутбук, але з невеликими модифікаціями:

- транспонування матриці виконувалось до циклу, а не в циклі;
- як критерій зупинки використовувалась комбінація із максимальної кількості ітерацій (1000) та обмеження на норму вектора;
- результат повертався як вектор 8-бітних цілих чисел (numpy.int8) для ефективного кешування.

Реалізація методу лінійного програмування була взята із бібліотеки SciPy [16], також із деякими модифікаціями:

- значення S найбільших за модулем координат встановлювались рівними 1, решта – 0;
- результат повертався як вектор 8-бітних цілих чисел ($numpy.int8$) для ефективного кешування та економного споживання пам’яті.

Важливим моментом у роботі з модулем лінійного програмування бібліотеки SciPy є вибір алгоритму для розв’язання задачі опуклої оптимізації. Бібліотека підтримує ряд таких методів, які дуже відрізняються за часом обчислень. Трьома ключовими алгоритмами є:

- симплекс-метод (*simplex*) [14] із вибором поворотних точок за правилом уникання циклів [17];
- метод внутрішніх точок (*interior-point*) [18];
- набір високопродуктивних солверів *HiGHS* [19].

В експериментах ці три методи давали однакові результати, втім HiGHS працював у середньому в 4 рази швидше, ніж interior-point, і в 8 разів швидше, ніж simplex, тому використовувався саме HiGHS.

Алгоритм 6 (робота пам'яті)

Крок 1. Ініціалізуємо CS-SDM (див. алгоритм 2).

Крок 2. Зчитуємо розріджені вектори $\{\vec{a}_i \mid 1 \le i \le I, \vec{a}_i \in \{0,1\}^L$ із диску.

Крок 3. Для кожного пакета $1 \le j_0 \le J$:

Крок 3.1. Перетворюємо кожен розріджений бінарний вектор довжини L пакета $A_{j_0} = \left\{\vec{a}_i \mid j_0 \cdot \frac{I}{J} < i \le (j_0+1) \cdot \frac{I}{J}\right\}$ за допомогою матриці $\mathbf{\Lambda}$ на цілочисельний вектор довжини L. Отримуємо пакет $A_{j_0}^{(CS)} = \left\{\mathbf{\Lambda} \cdot \vec{a}_i \mid j_0 \cdot \frac{I}{J} \le i \le (j_0+1) \cdot \frac{I}{J}\right\}$.

Крок 3.2. Записуємо кожен вектор пакета $A_{j_0}^{(CS)}$ і його збурений вектор до CS-SDM (див. алгоритм 3).

Крок 3.3. Зчитуємо із CS-SDM щільні сигнали по всіх вже записаних пакетах $A'_{j_0} = \bigcup_{j=1}^{j_0} A_j$ (див. алгоритм 4). Отримуємо множину M-розрядних дійсних векторів $V_{j_0} = \left\{\vec{v_i} \middle| \vec{v_i} \in \mathbb{R}^M, 1 \le i \le (j_0+1) \cdot \frac{I}{J}\right\}$. Також, для кожного отримуємо збурену адресу, міняючи одну випадково обрану 1 на 0, і читаємо за цією адресою із CS-SDM .

Крок 3.4. Покоординатно округляємо вектори із V_{j_0} і записуємо на диск.

Крок 4. Фіналізуємо CS-SDM (див. алгоритм 5).

Алгоритм 7 (відновлення розріджених векторів)

Крок 1. Зчитуємо випадкову матрицю $\mathbf{\Lambda}$.

Крок 2. Для кожного пакета $1 \le j \le J$:

Крок 2.1. Зчитуємо щільні сигнали пакета A'_j.

Крок 2.2. За допомогою одного із методів відновлень (CoSaMP, LinProg) отримуємо пакет розріджених бінарних векторів $\hat{A}'_j = \left\{\hat{\vec{a}}_i \mid j \cdot \frac{I}{J} < i \le (j+1) \cdot \frac{I}{J}\right\}$.

Крок 2.3. Зчитуємо оригінальні розріджені вектори A_j.

Крок 2.4. Рахуємо метрики оцінки $\hat{A}'_j$ відносно A_j.

Зауваження. Послідовність пакетів $\{A_j\}$, отримана в результаті роботи алгоритму 5, – зростаюча, а оскільки і CoSaMP, і LinProg – детерміновані алгоритми, то можна покращити алгоритм 6, додавши кешування результатів відновлення бінарних розріджених векторів.

Алгоритм 7* (відновлення розріджених векторів із кешуванням)

Крок 1. Зчитуємо випадкову матрицю $\mathbf{\Lambda}$.

Крок 2. Ініціалізуємо хеш-таблицю $cache$ (ключі – байт-строки дійсних векторів довжини M, значення – розріджені бінарні вектори довжини L).

Крок 3. Для кожного пакета $1 \le j \le J$:

Крок 3.1. Зчитуємо щільні сигнали пакета A'_j.

Крок 3.2. Для кожного вектора із пакета шукаємо розріджений відповідний вектор у cache. Якщо не знаходимо – відновлюємо за допомогою CoSaMP чи LinProg і записуємо в cache.

Крок 3.3. Зчитуємо оригінальні розріджені вектори A_j.

Крок 3.4. Рахуємо метрики оцінки $\hat{A}'_j$ відносно A_j.

Таким чином, всі описані алгоритми є природно паралельними. Переважна більшість їх операцій – локальна (задовільняє умовам дрібнозернистої геометричної декомпозиції), що створює умови для ефективної реалізації на GPU. Як платформа для реалізації була обрана NVIDIA CUDA.

Порівняно із іншими конструкціями (Джекела, Канерви), CS-SDM споживає пам'ять дуже економно, що дає змогу мати більше фізичних комірок і, відповідно, ширший простір адрес. Для класичних конструкцій ми мали розрядність SDM рівну довжині адрес ($M = L = 600$), водночас як для CS-SDM розрядність після стискання вдавалось без помітної втрати якості відновлення опустити до $6 \cdot s$, s – кількість одиниць у тестових векторах, $s \in \{12, 16, 20\}$. Тобто CS-SDM пропонує організацію у 5– 8 разів ефективнішу, ніж конструкцію Канерви та Джекела.

Експерименти проводилися на графічному прискорювачі GeForce RTX 2080 Ti (архітектура Turing, 11 ГБ пам'яті GDDR6, 4352 CUDA-ядер). Кількість фізичних комірок вибиралася так, щоб повністю заповнити пам'ять цього прискорювача. Для CS-SDM вона визначалась довжиною векторів, що записуються: $M = k \cdot s$. Тому, залежно від числа одиниць s та коефіцієнта k, кількість фізичних комірок змінювалась від 30 млн. до 100 млн. Для двох інших конструкцій (Канерви та Джекела) у пам'яті вмістилося по 15 млн. комірок розрядності $L = 600$.

На рисунку показано, що навіть за значного стискання застосування швидкого «жадібного» алгоритму CoSaMP дає якісні результати.

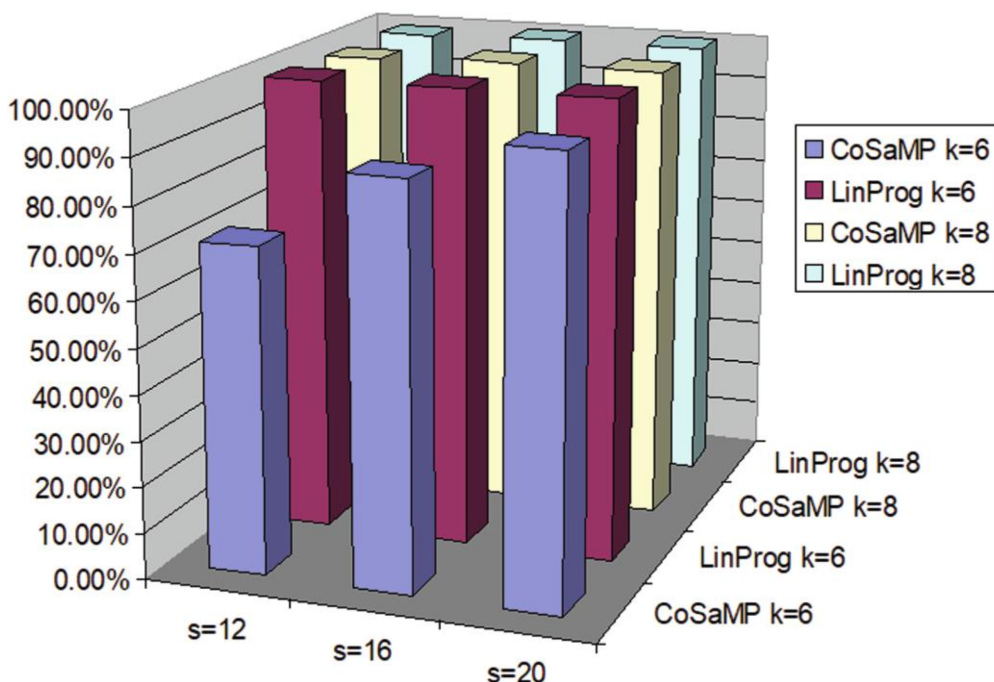

РИСУНОК. Відсоток коректно відновлених векторів із однією помилкою в адресі (s – кількість одиниць, K – рівень стиснення)

Висновки. Отримані результати показують, що конструкція CS-SDM – природно-паралельна і відповідає за будовою своїх алгоримів архітектурі систем з масивним паралелізмом. Проведені експерименти показали високу продуктивність розробленої реалізації блоку SDM (до 136 мкс на 1 запис, до 868 мкс на 1 читання з урахуванням кодування/декодування, до 255 мкс на 1 читання в рамках блоку SDM).

Буде досліджено ефективність алгоритмів при розв'язанні задач бінарних розподілених подань.

Список літератури

1. Kanerva P. Sparse Distributed Memory. Cambridge, MA: MIT Press, 1988. 180 p. https://isbnsearch.org/isbn/9780262111324
2. Jaeckel L.A. An Alternative Design for a Sparse Distributed Memory : Report RIACS TR 89.28 / Research Institute for Advanced Computer Science, NASA Ames Research Center. 1989. https://ntrs.nasa.gov/citations/19920001073
3. Albus J.S. A theory of cerebellar functions. *Mathematical Biosciences*. 1971. **10** (1-2). P. 25–61. https://doi.org/10.1016/0025-5564(71)90051-4
4. Smith D.J., Forrest S., Perelson A.S. Immunological memory is associative. *Artificial Immune Systems and Their Applications* / Dasgupta, D. (eds). Berlin, Heidelberg: Springer, 1999. P. 105–112. https://doi.org/10.1007/978-3-642-59901-9_6
5. Sjödin G. The Sparchunk Code: A method to build higher-level structures in a sparsely encoded SDM. *IJCNN/WCCI'98*: Proceedings of IEEE International Joint Conference on Neural Networks. London: Springer, 1998. P. 50–58. https://doi.org/10.1109/IJCNN.1998.685982
6. Rachkovskij D.A., Kussul E.M. Binding and normalization of binary sparse distributed representations by context-dependent thinning. *Neural Comput*. 2001. **13** (2). P. 411–452. https://doi.org/10.1162/089976601300014592
7. Schlegel K., Neubert P., Protzel, P. A comparison of vector symbolic architectures. *Artif Intell Rev*. 2022. **55**. P. 4523–4555. https://doi.org/10.1007/s10462-021-10110-3
8. Gayler R.W. Vector symbolic architectures are a viable alternative for Jackendoff's challenges. *Behavioral and Brain Sciences*. 2006. **29** (1). P. 78–79. https://doi.org/10.1017/S0140525X06309028
9. Candès E.J., Romberg J., Tao T. Stable signal recovery from incomplete and inaccurate measurements. *Comm. Pure Appl. Math.* 2006. **59** (8). P. 1207–1223. https://doi.org/10.1002/cpa.20124
10. Candès E.J., Wakin M.B. An introduction to compressive sampling. *IEEE Signal Process. Mag.* 2008. **25** (2). P. 21–30. https://doi.org/10.1109/MSP.2007.914731
11. Vdovychenko R., Tulchinsky V. Increasing the Semantic Storage Density of Sparse Distributed Memory. *Cybernetics and Systems Analysis*. 2022. **58** (3). P. 331–342. https://doi.org/10.1007/s10559-022-00465-y
12. Vdovychenko R., Tulchinsky V. Sparse Distributed Memory for Binary Sparse Distributed Representations. *ACM International Conference Proceeding Series* (ICMLT'2022: 7th International Conference on Machine Learning Technologies). 2022. P. 266–270. https://doi.org/10.1145/3529399.3529441
13. Needell D., Tropp J.A. CoSaMP: Iterative signal recovery from incomplete and inaccurate samples. *Applied and Computational Harmonic Analysis.* 2008. **26** (3). P. 301–321. https://doi.org/10.1016/j.acha.2008.07.002
14. Dantzig G.B. Linear programming and extensions. Princeton, NJ: Princeton University Press. 1963. 656 p. https://doi.org/10.7249/R366
15. Virmaux A. CoSaMP implementation in Python/NumPy. 2017. https://github.com/avirmaux/CoSaMP/blob/master/cosamp.ipynb (дата звернення: 14.09.2022).
16. Virtanen P., Gommers R., Oliphant T.E. et al. SciPy 1.0: fundamental algorithms for scientific computing in Python. *Nature Methods.* 2020. **17** (3). P. 261–272. https://doi.org/10.1038/s41592-019-0686-2
17. Bland R.G. New finite pivoting rules for the simplex method. *Mathematics of Operations Research.* 1977. **2** (2). P. 103–107. https://doi.org/10.1287/moor.2.2.103
18. Andersen E.D., Andersen K.D. The Mosek Interior Point Optimizer for Linear Programming: An Implementation of the Homogeneous Algorithm. *High Performance Optimization, Applied Optimization*, / Frenk, H., Roos, K., Terlaky, T., Zhang, S. (eds). Boston, MA: Springer, 2000. **33**. P. 197–232. https://doi.org/10.1007/978-1-4757-3216-0_8
19. Huangfu Q., Hall J.A. Parallelizing the dual revised simplex method. *Mathematical Programming Computation*. 2018. **10** (1). P. 119–142. https://doi.org/10.1007/s12532-017-0130-5

Одержано 14.09.2022

Вдовиченко Руслан Олександрович,
аспірант Інституту кібернетики імені В.М. Глушкова НАН України, Київ,
https://orcid.org/0000-0002-5929-6155
ruslan.vdovichenko1@gmail.com

Тульчинський Вадим Григорович,
доктор фізико-математичних наук, старший науковий співробітник, завідувач відділу автоматизації програмування Інституту кібернетики імені В.М. Глушкова НАН України, Київ.
https://orcid.org/0000-0002-0280-223X

UDC 519.6:004.8

Ruslan Vdovychenko [*], Vadim Tulchinsky

Parallel Implementation of Sparse Distributed Memory for Semantic Storage

V.M. Glushkov Institute of Cybernetics of the NAS of Ukraine, Kyiv
[*] *Correspondence: ruslan.vdovichenko1@gmail.com*

Introduction. Sparse Distributed Memory (SDM) and Binary Sparse Distributed Representations (Binary Sparse Distributed Representations, BSDR), as two phenomenological approaches to biological memory modelling, have many similarities. The idea of their integration into a hybrid semantic storage model with SDM as a low-level cleaning memory (brain cells) for BSDR, which is used as an encoder of high-level symbolic information, is natural. A hybrid semantic store should be able to store holistic data (for example, structures of interconnected and sequential key-value pairs) in a neural network. A similar design has been proposed several times since the 1990s. However, the previously proposed models are impractical due to insufficient scalability and/or low storage density. The gap between SDM and BSDR can be bridged by the results of a third theory related to sparse signals: Compressive Sensing or Sampling (CS). In this article, we focus on the highly efficient parallel implementation of the CS-SDM hybrid memory model for graphics processing units on the NVIDIA CUDA platform, analyze the computational complexity of CS-SDM operations for the case of parallel implementation, and offer optimization techniques for conducting experiments with big sequential batches of vectors.

The purpose of the paper is to propose an efficient software implementation of sparse-distributed memory for preserving semantics on modern graphics processing units.

Results. Parallel algorithms for CS-SDM operations are proposed, their computational complexity is estimated, and a parallel implementation of the CS-SDM hybrid semantic store is given. Optimization of vector reconstruction for experiments with sequential data batches is proposed.

Conclusions. The obtained results show that the design of CS-SDM is naturally parallel and that its algorithms are by design compatible with the architecture of systems with massive parallelism. The conducted experiments showed high performance of the developed implementation of the SDM memory block.

Keywords: GPU, CUDA, neural network, Sparse Distributed Memory, associative memory, Compressive Sensing.