Програмування інтелектуальних інформаційних систем

3 курс, осінь 2021

- Доц. Баклан І.В.
- Email: iaa@ukr.net
- Web: baklaniv.at.ua

Лекція 15

Triton: проміжна мова та компілятор для обчислень із плитковими нейронними мережами

15.1. Вступ

Ця лекція буде присвячена продовженню ознайомлення з мовою програмування Triton. Перевірка та розгортання нових дослідницьких ідей у сфері глибокого навчання часто обмежені наявністю ефективних обчислювальних ядер для певних базових примітивів. Зокрема, операції, які не можуть використовувати існуючі бібліотеки постачальників (наприклад, cuBLAS, cuDNN), ризикують зіткнутися з поганим використанням пристрою, якщо спеціальні реалізації не будуть написані експертами – зазвичай за рахунок портативності.

З цієї причини розробка нових абстракцій програмування для визначення користувацьких робочих навантажень глибокого навчання з мінімальними витратами на продуктивність стала вирішальною.

Ми представляємо Triton, мову та компілятор, зосереджений на концепції плитки, тобто багатовимірних підмасивів статичної форми. Наш підхід обертається навколо (1) мови на основі C і проміжного представлення (IR) на основі LLVM для вираження тензорних програм у термінах операцій над параметричними змінними мозаїки та (2) набору нових ходів оптимізації на рівні плитки для їх компіляції. програми в ефективний код GPU. Ми демонструємо, як Triton можна використовувати для створення портативних реалізацій ядер множення матриць і згортки нарівні з налаштованими вручну бібліотеками постачальників (cuBLAS / cuDNN) або для ефективної реалізації останніх дослідницьких ідей, таких як згортки зсуву.

Нещодавнє відродження глибоких нейронних мереж (DNN) значною мірою сприяло [24] широкій доступності програмованих паралельних обчислювальних пристроїв. Зокрема, постійне покращення продуктивності багатоядерних архітектур (наприклад, графічних процесорів) відіграло фундаментальну роль, дозволяючи дослідникам та інженерам досліджувати все більший спектр моделей, що стають більш великими, використовуючи все більше і більше даних. Ці зусилля були підтримані колекцією бібліотек постачальників (cuBLAS, cuDNN), спрямованих на те, щоб якомога швидше донести до практиків останні інновації обладнання. На жаль, ці бібліотеки підтримують лише обмежений набір тензорних операцій, залишаючи реалізацію нових примітивів експертам [13, 17, 25].

Це спостереження призвело до розробки різних домено-специфічних мов (DSL) для DNN, заснованих на поліедричних механізмах (наприклад, Tensor Comprehensions [43]) та/або методах синтезу циклів (наприклад, Halide [37], TVM [10]). та PlaidML[22]). Але хоча ці системи, як правило, добре працюють для певних класів проблем, таких як згортки, що розділяються по глибині (наприклад, MobileNet [20]), на практиці вони часто набагато повільніші, ніж бібліотеки постачальників (див., наприклад, рис.15.1), і їм не вистачає виразності. необхідні для реалізації структурованих шаблонів розрідженості [28, 31, 47], які не можуть бути безпосередньо визначені за допомогою індексів афінного масиву у вкладених циклах.

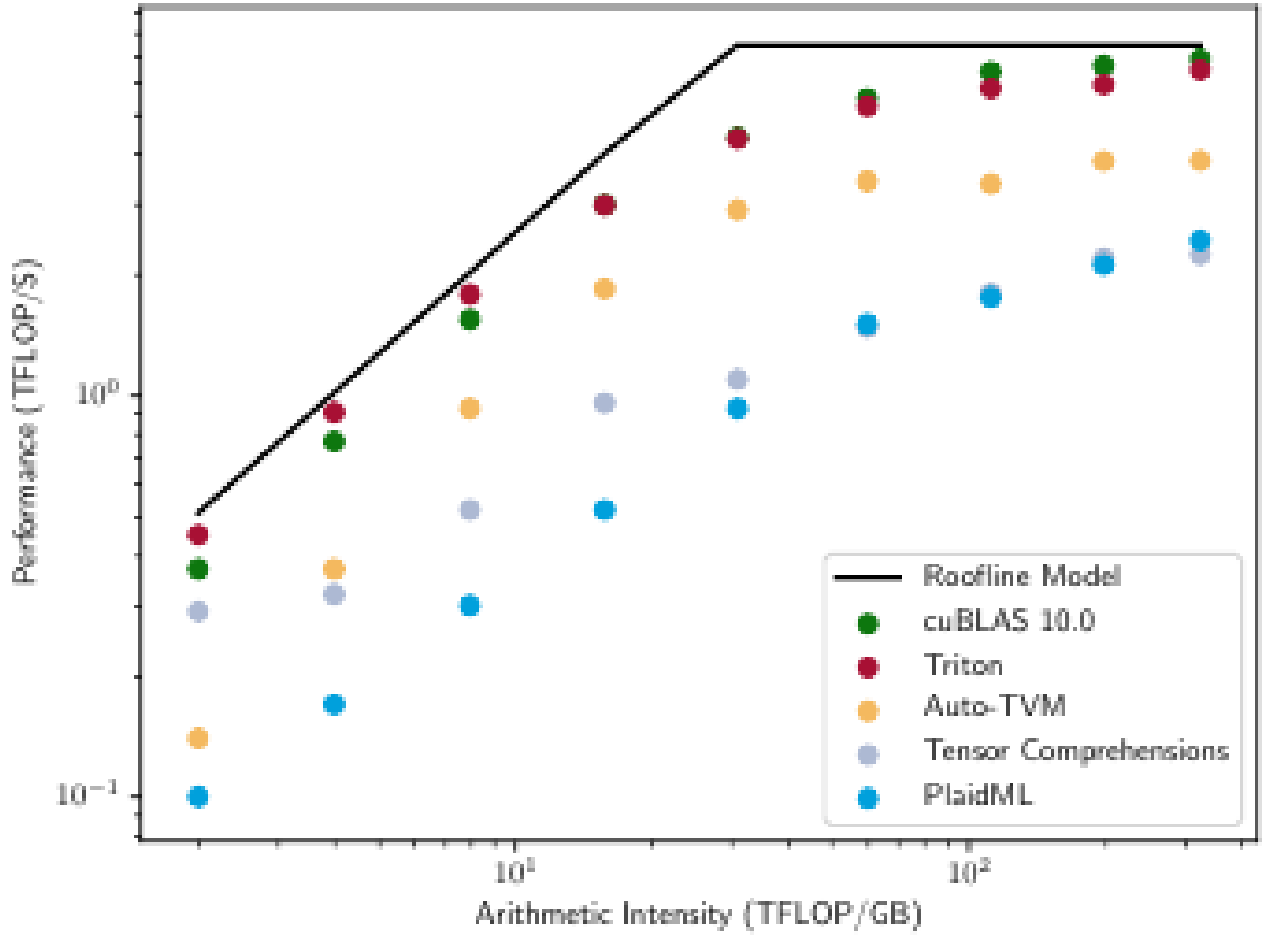


Рис.15.1. Продуктивність різних реалізацій моделі $C = AB^T$ проти Roofline [46] (NVIDIA GeForce GTX1070), де $A \in R^{1760\times1760}$ і $B \in R^{N\times1760}$ як N модулює арифметичну інтенсивність.

Ці проблеми часто вирішуються шляхом використання мікроядер [11, 21] – тобто рукописних елементів на рівні плитки – але це рішення вимагає великої кількості ручної праці та не має портативності. І хоча нещодавно було запропоновано кілька високорівневих абстракцій програмування для розкладки [23, 41], базові серверні системи компілятора все ще не підтримують операції та оптимізації на рівні плитки. З цією метою ми представляємо Triton (рис. 15.2), проміжну мову з відкритим вихідним кодом і компілятор для визначення та компіляції плиткових програм в ефективний код GPU.

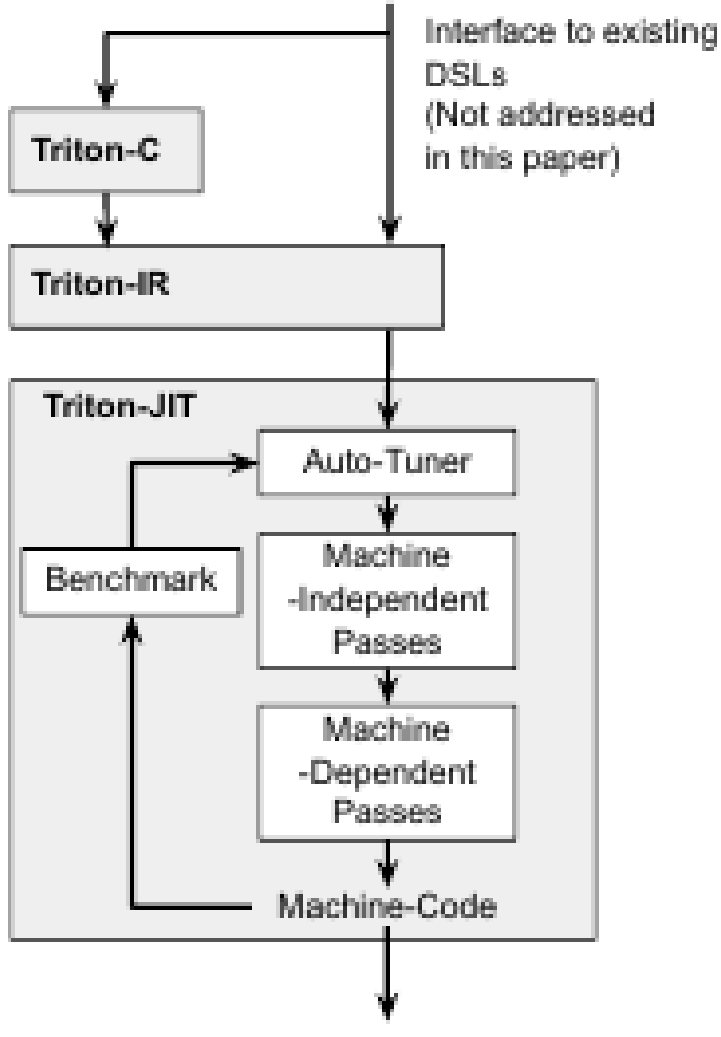


Рис.15.2. Погляд на Triton

Основні внески цієї статті підсумовані таким чином:

• Triton-C (Розділ 15.3): C-подібна мова для вираження тензорних програм у термінах параметричних змінних плитки. Метою цієї мови є забезпечення стабільного інтерфейсу для існуючих транскомпіляторів DNN (наприклад, PlaidML, Tensor Comprehensions) і програмістів, знайомих з CUDA. Листинг 15.1 показує вихідний код Triton-C, пов’язаний із простим завданням на множення матриці.

• Triton-IR (Розділ 15.4): Проміжне представлення (IR) на основі LLVM, яке забезпечує середовище, придатне для аналізу, трансформації та оптимізації програм на рівні плитки. У листингу 15.5 показано код Triton-IR для функції Rectified Linear Unit (ReLU). Тут програми Triton-IR створюються безпосередньо з Triton-C під час аналізу, але автоматичне генерування з вбудованих DSL або компіляторів DNN вищого рівня (наприклад, TVM) також може бути досліджено в майбутньому.

• Triton-JIT (Розділ 15.5): Компілятор Just-In-Time (JIT) і бекенд генерації коду для компіляції програм Triton-IR в ефективний біт-код LLVM. Це включає (1) набір машинно-незалежних проходів на рівні плитки, спрямованих на спрощення вхідних обчислювальних ядер незалежно від будь-якої мети компіляції; (2) набір машинно-залежних проходів на рівні плитки для створення ефективних готових до GPU LLVM-IR; і (3) автонастроювач, який оптимізує будь-який мета-параметр, пов'язаний з наведеними вище проходами.

• Числові експерименти (Розділ 15.6): чисельне оцінювання Triton, яке демонструє його здатність (1) генерувати реалізації множення матриці нарівні з cuBLAS і до 3 разів швидше, ніж альтернативні DSL в рекуррентних і трансформаторних нейронних мережах; (2) повторно реалізувати алгоритм IMPLICIT_GEMM cuDNN для щільної згортки без втрати продуктивності; і (3) створювати ефективні реалізації нових дослідницьких ідей, таких як модулі shift-conv [47].

Перед цим документом буде подано короткий аналіз наявної відповідної літератури (Розділ 15.2), а на завершення – резюме роботи (Розділ 15.7).

Супутні напрямки роботи

Існування фреймворків [1, 9, 36] та бібліотек для глибокого навчання було критичним для появи нових архітектур і алгоритмів нейронних мереж. Але незважаючи на досягнення аналітичної [5, 48] та емпіричної [6, 30] евристики для компіляторів лінійної алгебри, це програмне забезпечення все ще покладається на підпрограми, оптимізовані вручну (наприклад, cuBLAS і cuDNN). Це призвело до розробки різних DSL і компіляторів для DNN, які зазвичай базуються на одному з трьох різних підходів:

• IP тензорного рівня використовували XLA [16] та Glow [38] для перетворення тензорних програм у попередньо визначені шаблони операцій LLVM-IR та CUDA-C (наприклад, тензорні скорочення, поелементні операції тощо) за допомогою шаблону відповідність.
• Поліедральна модель [18] була використана Tensor Comprehensions (TC) [43] і Diesel [14] для параметризації та автоматизації компіляції одного або багатьох шарів DNN у програми LLVM-IR та CUDA-C.
• Синтезатори циклів використовувалися Halide [37] та TVM [10] для перетворення тензорних обчислень у гнізда циклів, які можна вручну оптимізувати за допомогою визначених користувачем (хоча, можливо, параметричних [11]) розкладів.

Навпаки, Triton покладається на додавання операцій на рівні плитки та оптимізації в традиційні конвеєри компіляції. Цей підхід забезпечує (1) більшу гнучкість, ніж XLA і Glow; (2) підтримка неафінних тензорних індексів на відміну від TC та Diesel; і (3) автоматичний висновок про можливий графік виконання, який інакше довелося б вручну вказати для галогеніду або TVM. Переваги Triton забезпечуються ціною збільшення зусиль у програмуванні – див. Лістінг 15.2 щодо реалізацій множення матриці в цих DSL.

```
C = tf.matmul(A, tf.transpose(B))      // TF
C[i, j: I, J] = +(A[i, k] * B[j, k]); // PlaidML
C(i, j) +=! A(i, k) * B(j, k)          // TC
tvm.sum(A[i, k] * B[j, k], axis=k)     // TVM
```

Лістінг 15.2. $C = A \times B^T$ в TF, PlaidML, TC та TVM

15.3 Мова Triton-C

Метою Triton-C є забезпечення стабільного інтерфейсу для існуючих (і майбутніх) транскомпіляторів DNN, а також програмістів, знайомих з низькорівневим програмуванням GPU. У цій частині лекції ми описуємо CUDA-подібний синтаксис Triton-C (розділ 15.3.1), його семантику, подібну до Numpy[35] (розділ 15.3.2) і його модель програмування - «одна програма, множина даних» (SPMD) (розділ 15.3.3).

15.3.1 Синтаксис

Синтаксис Triton-C заснований на синтаксисі ANSI C (і точніше CUDA-C), але був змінений і розширений (див. Лістінг 15.3), щоб вмістити семантику та модель програмування, описану в двох наступних підрозділах. Ці зміни поділяються на такі категорії:

Оголошення плиток: ми додали спеціальний синтаксис для оголошення багатовимірних масивів (наприклад, `int tile[16, 16]`), щоб підкреслити їх семантичну відмінність від вкладених масивів, знайдених у ANSI C (наприклад, `int tile[16][16]`). Форми плитки мають бути постійними, але їх також можна зробити параметричними за допомогою ключового слова настроюваний. Одновимірні цілі плитки можна ініціалізувати за допомогою еліпсів (наприклад, `int range[8] = 0 ... 8`).

```
// Broadcasting semantics
slice          : ':' | 'newaxis'
slice_list     : slice | slice_list ',' slice
slice_expr     : postfix_expr | expr '[' slice_list ']'
// Range initialization
constant_range : expr '...' expr
// Intrinsics
global_range   : 'get_global_range' '(' constant ')'
dot            : 'dot' '('  expr ',' expr ')'
trans          : 'trans' '(' expr ',' expr ')'
intrinsic_expr : global_range | dot | trans
// Predication
predicate_expr : '@' expr
// Tile extensions for abstract declarators
abstract_decl  : abstract_decl | '[' constant_list ']'
// Extensions of C expressions
expr           :  expr | constant_range  |  slice_expr
               | intrinsic_expr
// Extensions of C specifiers
storage_spec : storage_spec | 'kernel'
type_spec    : type_spec | 'tunable'
// Extensions of C statements
statement    : statement | predicate_expr statement
```

Лістінг 15.3. Граматичні розширення для Triton-C. Ми припускаємо існування певних конструкцій C, показаних синім кольором.

Вбудована функція: у той час як загальний синтаксис C був збережений для поелементних операцій з масивами (`+`, `-`, `&&`, `*` тощо), різні вбудовані функції (`dot`, `trans`, `get_global_range`) були додані для підтримки семантики плитки (розділ 15.3.2.1) і модель програмування SPMD.

Трансляція: N-вимірні фрагменти можна транслювати вздовж будь-якої конкретної осі за допомогою ключового слова `newaxis` і звичайного синтаксису зрізів (наприклад, `int broadcast [8, 8] = range[:, newaxis]` для накладання стовпців). Зауважте, що розрізання плиток для отримання скалярів або підмасивів в іншому випадку заборонено.

Предикація: основний потік керування в операціях з плитками (Розділ 15.4.3) досягається за допомогою використання предикативних операторів за допомогою префікса «@».

15.3.2 Семантика

15.3.2.1 Семантика плитки

Існування вбудованих типів плиток і операцій (тобто семантики плиток) у Triton-C пропонує дві основні переваги. По-перше, він спрощує структуру тензорних програм, приховуючи важливі деталі продуктивності, що стосуються об'єднання внутрішньоплиткової пам'яті [12], керування кеш-пам'яттю [32] та використання спеціалізованого обладнання [27]. По-друге, це відкриває двері для компіляторів для автоматичного виконання цих оптимізацій, як обговорювалося в розділі 15.5.

15.3.2.2 Семантика мовлення

Плитки в Triton-C суворо типізовані в тому сенсі, що деякі інструкції статично вимагають, щоб їхні операнди підкорялися строгим обмеженням форми. Наприклад, скаляр не може бути доданий до масиву, якщо він спочатку не буде належним чином переданий.

Семантика широкомовлення [35] надає набір правил для неявного виконання цих перетворень (приклад дивіться в листингу 15.4):

1. *Заповнення*: форма найкоротшого операнда доповнюється ліворуч одиницями, поки обидва операнди не будуть мати однакову розмірність.
2. *Трансляція*: вміст обох операндів реплікується стільки разів, скільки потрібно, доки їх форма не стане ідентичною; якщо це неможливо зробити, видається помилка.

```
int a[16], b[32, 16], c[16, 1];
// a is first reshaped to [1, 16]
// and then broadcast to [32, 16]
int x_1[32, 16] = a[newaxis, :] + b;
// Same as above but implicitly
int x_2[32, 16] = a + b;
// a is first reshaped to [1, 16]
// a is broadcast to [16, 16]
// c is broadcast to [16, 16]
int y[16, 16] = a + c;
```

Лістінг 15.4. Семантика мовлення на практиці

15.3.3 Модель програмування

Виконання коду CUDA [33] на графічних процесорах підтримується моделлю програмування SPMD [4], в якій кожне ядро пов'язане з ідентифікованим блоком потоків у так званій стартовій сітці. Модель програмування Triton подібна, але кожне ядро є однопотоковим (хоча автоматично розпаралелюваним) і пов’язаним з набором глобальних діапазонів, який варіюється від екземпляра до екземпляра (див. Малюнок 15.3). Такий підхід призводить до більш простих ядер, у яких не існує примітивів, подібних до CUDA (синхронізація спільної пам’яті, міжпотоковий зв’язок тощо).

Глобальні діапазони, пов’язані з ядром, можна запитати за допомогою вбудованої функції `get_global_range(axis)`, щоб створити, наприклад, плитки покажчиків, як показано в листингу 15.1.

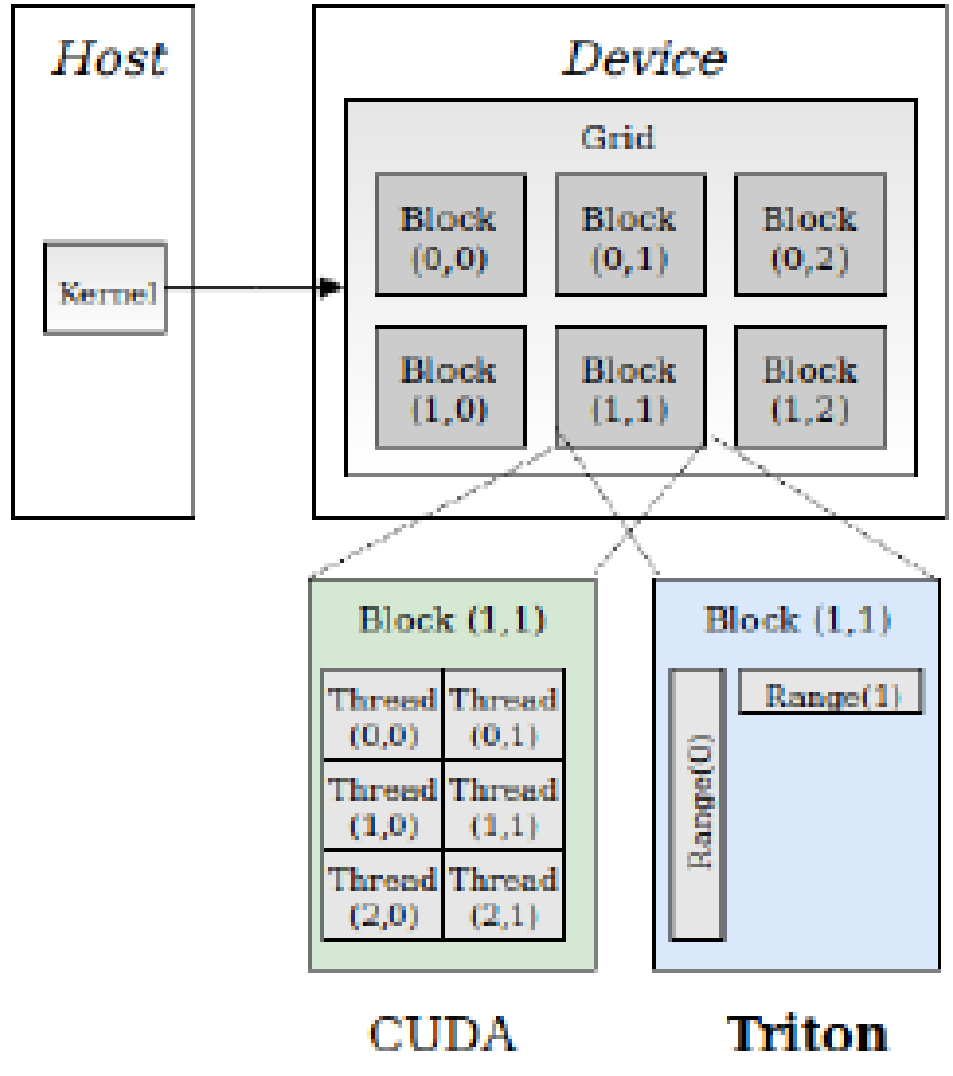


Рис.15.3. Різниця між моделлю програмування CUDA та Triton.

15.4 Triton-IR

Triton-IR — це проміжне представлення (IR), засноване на LLVM, метою якого є забезпечення середовища, придатного для аналізу, трансформації та оптимізації програм на рівні плитки. У цій лекції програми Triton-IR створюються безпосередньо з Triton-C під час синтаксичного аналізу, хоча в майбутньому вони також можуть бути створені безпосередньо з DSL вищого рівня.

Програми Triton-IR і LLVM-IR мають однакову структуру високого рівня (згадується в Розділі 15.4.1), але перша також включає ряд розширень, необхідних для потоку даних на рівні плитки (Розділ 15.4.2) і потоку керування (Розділ 15.4.3) аналіз. Ці нові розширення мають вирішальне значення для виконання оптимізацій, описаних у розділі 15.5, і для безпечного доступу до тензорів довільних форм, як показано в розділі 15.6.

15.4.1 Структура

15.4.1.1 Модулі

На найвищому рівні програми Triton-IR складаються з одного або кількох базових блоків компіляції, відомих як модулі. Ці модулі компілюються незалежно один від одного і в кінцевому підсумку агрегуються компонувальником, роль якого полягає в тому, щоб розв’язувати прямі оголошення та адекватно об’єднувати глобальні визначення.

Кожен модуль сам по собі складається з функцій, глобальних змінних, констант та інших різноманітних символів (наприклад, метаданих, атрибутів функцій).

15.4.1.2 Функції

Визначення функції Triton-IR складаються з типу повернення, імені та потенційно порожнього списку аргументів. За бажанням можна додати додаткові специфікатори видимості, вирівнювання та зв’язків. Атрибути функцій (такі як підказки для вбудовування) та атрибути параметрів (наприклад, підказки лише для читання, підказки псевдонімів) також можна вказати, що дозволить бекендам компілятора виконувати більш агресивну оптимізацію, наприклад, краще використовуючи кеш пам’яті лише для читання.

Після цього заголовка йде тіло, що складається зі списку базових блоків, взаємозалежності яких утворюють графік потоку керування (CFG) функції.

15.4.1.3 Основні блоки

Базові блоки, за визначенням, — це прямолінійні кодові послідовності, які можуть містити лише так звані інструкції термінатора (тобто розгалуження, повернення) на своєму кінці.
Triton-IR використовує форму Static Single Assignment (SSA), що означає, що кожна змінна в кожному базовому блоці має бути (1) призначена лише один раз і (2) визначена перед використанням. При цьому кожен базовий блок неявно визначає графік потоку даних (DFG), різні шляхи якого відповідають ланцюгам use-def у представленні програми SSA. Цю форму можна створити безпосередньо з абстрактних синтаксичних дерев (AST), як показано в [7].

15.4.2 Підтримка аналізу потоків даних на рівні плитки

15.4.2.1 Типи

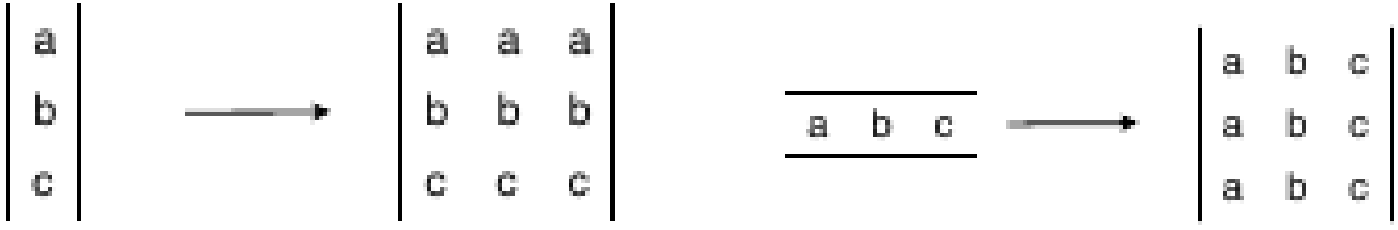
Багатовимірні плитки знаходяться в центрі аналізу потоків даних у Triton-IR і можуть бути оголошені за допомогою синтаксису, подібного до оголошення векторів у LLVM-IR. Наприклад, `i32<8, 8>` є типом, що відповідає 8 × 8 32-бітним цілочисельним плиткам. Зауважте, що в Triton-IR немає настроюваного ключового слова, тому параметричні значення форми повинні бути розв’язані до створення програм. У нашому випадку це робиться автонастройщиком Triton-JIT (Розділ 15.5.3).

15.4.2.2 Інструкції

Triton-IR представляє набір інструкцій переміщення, метою яких є підтримка семантики мовлення, як описано в Розділі 15.3.2.2:

• Інструкція зміни форми створює плитку із зазначеними фігурами, використовуючи дані зі свого вхідного аргументу. Це особливо корисно для повторної інтерпретації змінних як масивів вищих розмірів, доповнюючи їх вхідні форми одиницями для підготовки неявного або явного мовлення.

• Інструкція широкомовної передачі створює плитку вказаних форм шляхом повторення її вхідного аргументу необхідну кількість разів уздовж розмірів розміру 1 – як показано на малюнку 15.4.



(a) [3 × 1] input (b) [1 × 3] input

Рис. 15.4. broadcast інструкція <3,3>

Звичайні скалярні інструкції (**`cmp`**, **`getelementptr`**, **`add`**, **`load`**...) були збережені та розширені для позначення поелементних операцій над операндами плитки. Нарешті, Triton-IR також надає спеціалізовані арифметичні інструкції для транспозицій (**`trans`**) і множення матриці (**`dot`**).

15.4.3 Підтримка аналізу потоку керування на рівні плитки

Однією з проблем, яка виникає через існування операцій на рівні плитки в Triton-IR, є невиразність потоку дивергентного керування всередині плиток. Наприклад, програмі може знадобитися частково захистити навантаження на рівні плитки від порушень доступу до пам’яті, але цього неможливо досягти за допомогою розгалуження, оскільки елементи плитки не можуть бути доступні окремо.

Ми пропонуємо вирішити це питання за допомогою використання предикативної форми SSA (PSSA) [8] та **ψ** - функцій [39]. Це вимагає додавання двох класів інструкцій (див. листинг 6) до Triton-IR:

- Інструкції cmpp [8] подібні до звичайних інструкцій порівняння (**`cmp`**), за винятком того факту, що вони повертають два протилежних предикати замість одного.
- Інструкція **`psi`** об'єднує інструкції з різних потоків передбачуваних інструкцій.

```
;pt[i,j], pf[i,j] = (true, false) if x[i,j] < 5
;pt[i,j], pf[i,j] = (false, true) if x[i,j] >= 5
%pt, %pf = icmpp slt %x, 5
@%pt %x1 = add %y, 1
@%pf %x2 = sub %y, 1
; merge values from different predicates
%x = psi i32<8,8> [%pt, %x1], [%pf, %x2]
%z = mul i32<8,8> %x, 2
```

Лістінг 15.6. Предикація рівня плитки в Triton-IR

15.5 Компілятор Triton-JIT

Метою Triton-JIT є спрощення та компіляція програм Triton-IR в ефективний машинний код за допомогою набору машинно-незалежних (Розділ 15.5.1) і машинозалежних (Розділ 15.5.2 проходів, що підтримуються механізмом автоматичного налаштування (Розділ 15.5.3).

15.5.1 Незалежні від машини проходи

15.5.1.1 Попередня вибірка

Робота з пам’яттю на рівні плитки всередині циклів може бути проблематичною, оскільки вони можуть викликати серйозні затримки, які неможливо приховати за відсутності достатньої кількості незалежних інструкцій.

Однак можна пом’якшити цю проблему безпосередньо в Triton-IR шляхом виявлення циклів і додавання відповідного коду попередньої вибірки, де це необхідно (див. Листинг 15.7).

```
B0:
 %p0 = getelementptr %1, %2
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = load %p
 ; increment pointer
 %p1 = getelementptr %p, %3
```

```
B0:
 %p0 = getelementptr %1, %2
 %x0 = load %p0
B1:
 %p = phi [%p0,B0], [%p1,B1]
 %x = phi [%x0,B0], [%x1,B1]
 ; increment pointer
 %p1 = getelementptr %p, %3
 ; prefetching
 %x1 = load %p
```

Лістінг 15.7. Автоматична попередня вибірка

15.5.1.2 Оптимізація вічка на рівні плитки

Наявність операцій на рівні плитки в Triton-IR відкриває нові можливості для оптимізаторів вічка [29]. Наприклад, ланцюжки транспозицій можна спростити, використовуючи тотожність $X=(X^T)^T$ для будь-якої плитки X. Ми вважаємо, що інші алгебраїчні властивості, пов’язані, наприклад, з діагональними плитками, також можуть бути використані в майбутньому.

15.5.2 Машинозалежні проходи

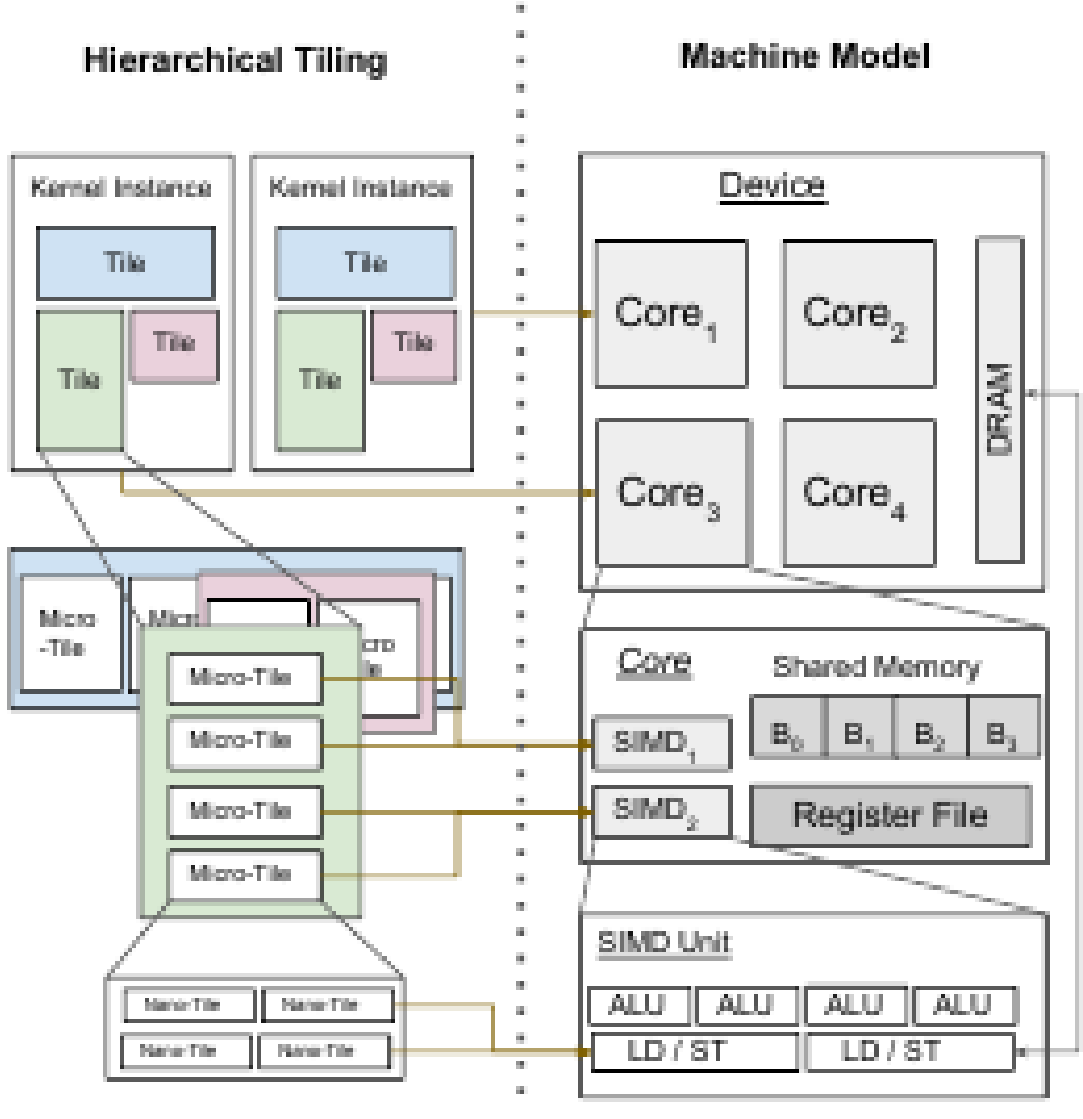
Зараз ми представляємо набір проходів оптимізації для машин які відповідають моделі високого рівня, показаній на малюнку 15.5.

В основному, оптимізації, виконані Triton-JIT, складаються з (1) ієрархічної плитки, (2) об’єднання пам’яті, (3) спільної виділення пам'яті та (4) синхронізація спільної пам'яті.

15.5.2.1 Ієрархічна плитка

Стратегії вкладених плиток (див. Рис.15.5) спрямовані на розкладання плитки в мікро-плитки і, зрештою, нано-плитки, щоб підігнати обчислювальні можливості машини та ієрархія пам'яті як якомога щільніше. Хоча ця техніка зазвичай використовується в системах автоматичного налаштування [34, 40], структура Triton-IR дає змогу автоматично перераховувати й оптимізувати дійсні конфігурації вкладених мозаїк для будь-якої програми, що виражається (і без необхідності багатогранної техніки).

Рис.15.5. Ієрархічна плитка в моделі машини Triton-IR



15.5.2.2 Об’єднання пам’яті

Вважається, що доступ до пам’яті об’єднується, коли сусідні потоки одночасно звертаються до сусідніх місць пам’яті. Це важливо, оскільки пам’ять зазвичай витягується великими блоками з DRAM.

Оскільки програми Triton-IR є однопоточними і автоматично розпаралелюються, наш бекенд компілятора може впорядковувати потоки всередині кожної мікро-плитки, щоб уникнути доступу до пам’яті, коли це можливо. Ця стратегія зменшує кількість транзакцій пам'яті, необхідних для завантаження стовпця плитки (див. Рис.15.6).

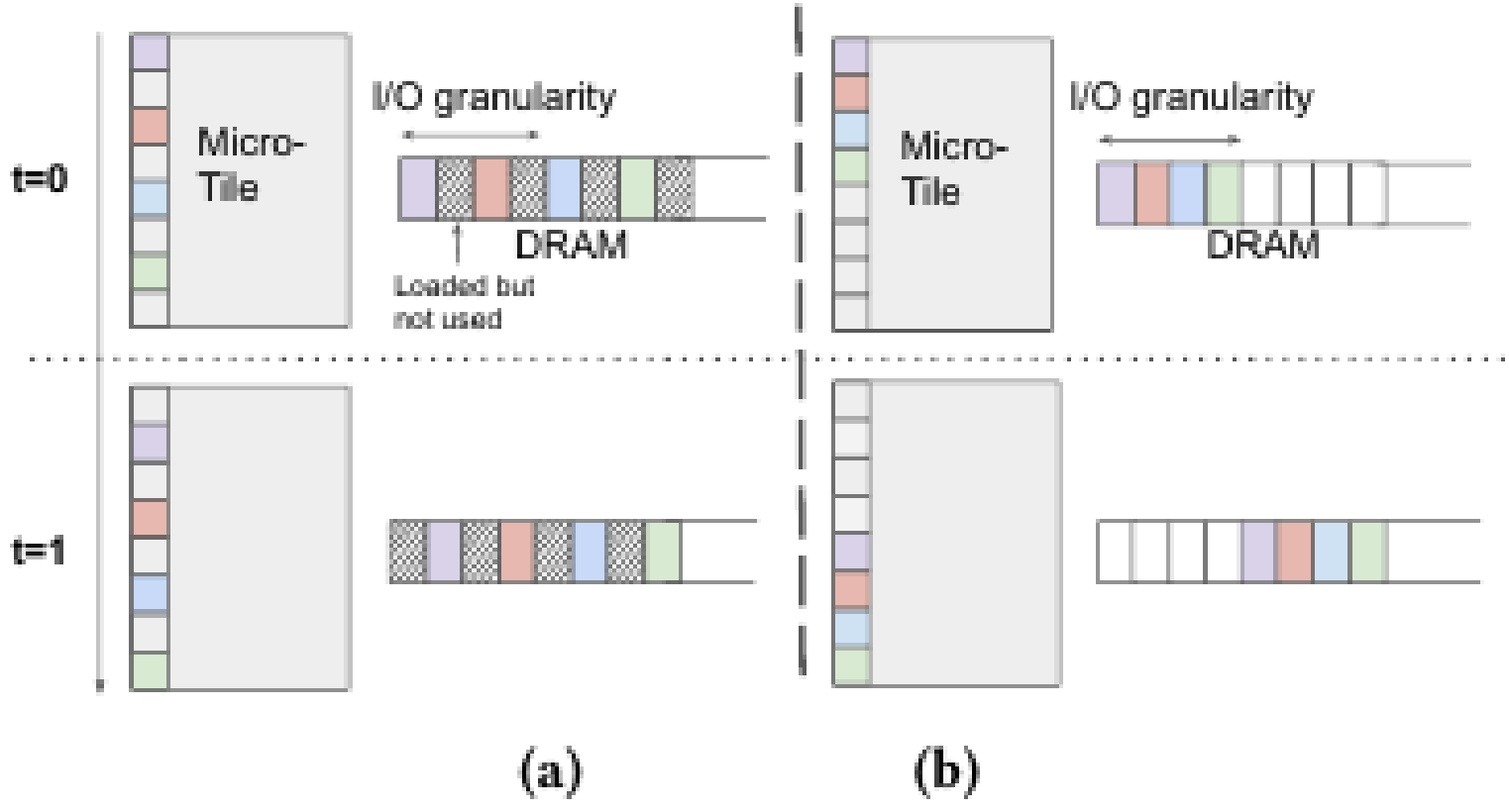


Рис.15.6. Необ’єднані (a) та об’єднані (b) доступи до DRAM. Різні нитки показані різними кольорами.

15.5.2.3 Виділення спільної пам'яті

Операції на рівні плитки, які мають високу інтенсивність арифметики (наприклад, точка), можуть отримати вигоду від тимчасового зберігання своїх операндів у швидкій спільній пам’яті. Мета проходу розподілу спільної пам’яті полягає в тому, щоб визначити, коли і де плитка повинна бути прихована в цьому просторі. Це можна зробити, як показано на рисунку 15.7, спочатку обчисливши живий діапазон кожної цікавить змінної, а потім використавши алгоритм розподілу пам’яті за лінійним часом, запропонований у [15].

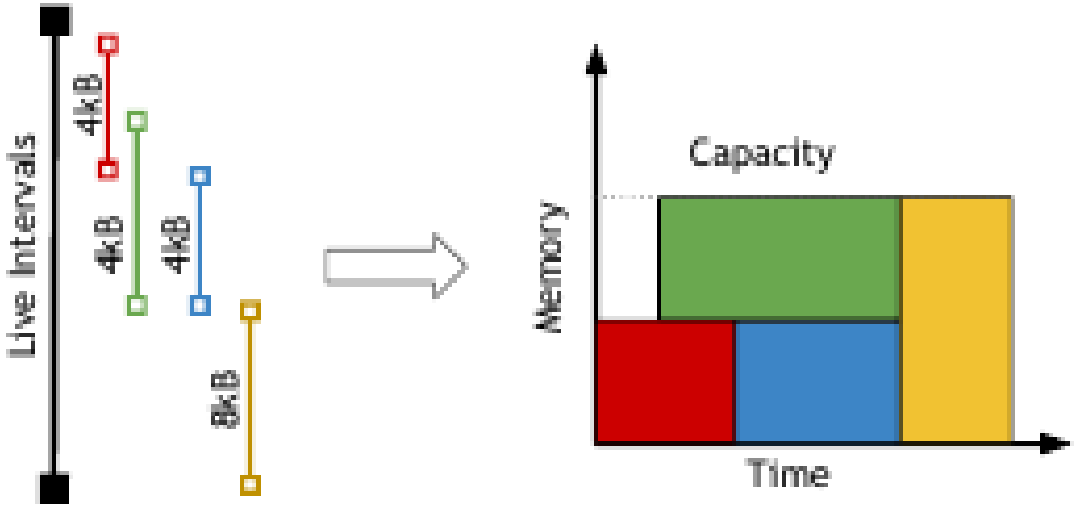


Рис.15.7. Спільне виділення пам'яті

15.5.2.4 Синхронізація спільної пам'яті

Зчитування та запис у спільну пам’ять у нашій моделі машини асинхронні. Мета проходу синхронізації спільної пам’яті автоматично вставляє бар’єри у згенерований вихідний код GPU, щоб зберегти коректність програми. Це робиться шляхом виявлення небезпек читання після запису (RAW) і запису після читання (WAR) за допомогою прямого аналізу потоку даних з такими рівняннями потоку даних:

$$in_s^{(RAW)} = \bigcup_{p \in \text{pred}(s)} out_p^{(RAW)}$$

$$in_s^{(WAR)} = \bigcup_{p \in \text{pred}(s)} out_p^{(WAR)}$$

$$out_s^{(RAW)} = \begin{cases} \emptyset & \textbf{if } in_s^{(RAW)} \cap read(s) \neq \emptyset \text{ (barrier)} \\ in_s^{(RAW)} \cup write(s) & \textbf{otherwise} \end{cases}$$

$$out_s^{(WAR)} = \begin{cases} \emptyset & \textbf{if } in_s^{(WAR)} \cap write(s) \neq \emptyset \text{ (barrier)} \\ in_s^{(WAR)} \cup read(s) & \textbf{otherwise} \end{cases}$$

15.5.3 Автотюнер

Традиційні автонастроювачі [42, 45] зазвичай покладаються на рукописні параметризовані шаблони коду для досягнення хорошої продуктивності на попередньо визначених робочих навантаженнях. На відміну від цього, Triton-JIT може витягувати оптимізаційні простори безпосередньо з програм Triton-IR, просто об'єднуючи мета-параметри, пов'язані з кожним із вищезазначених проходів оптимізації.

У цій лекції розглядається лише прохід ієрархічної плитки, що призводить до не більше ніж 3 параметрів розбиття на вимір на плитку. Ці параметри потім оптимізуються за допомогою вичерпного пошуку за степенями двійки між (a) від 32 до 128 для розмірів плитки; (b) 8 і 32 для розмірів мікроплитки; і (c) 1 і 4 для розмірів нано плитки. У майбутньому можна було б використовувати кращі методи автооб'єднання.

15.6 Чисельні експерименти

У цьому розділі ми оцінюємо продуктивність Triton на різних робочих навантаженнях з літератури Deep Learning. Ми використали NVIDIA GeForce GTX1070 і порівняли нашу систему з найновішими бібліотеками постачальників (cuBLAS 10.0, cuDNN 7.0), а також з пов’язаною технологією компілятора (Auto-TVM, TC, PlaidML). Якщо це можливо, ми автоматично налаштували ці DSL для кожного окремого розміру проблеми, дотримуючись вказівок офіційної документації.

15.6.1 Матричне множення

Завдання на множення матриці вигляду: $A = D \times W^T$ ($D \in R^{M \times K}$, $W \in R^{N \times K}$) лежать в основі обчислень нейронної мережі. Тут ми розглядаємо різноманітні завдання з рекурентних (DeepSpeech2 [3]) і трансформаторних [44] нейронних мереж; ми повідомляємо їх продуктивність на рис. 15.8.

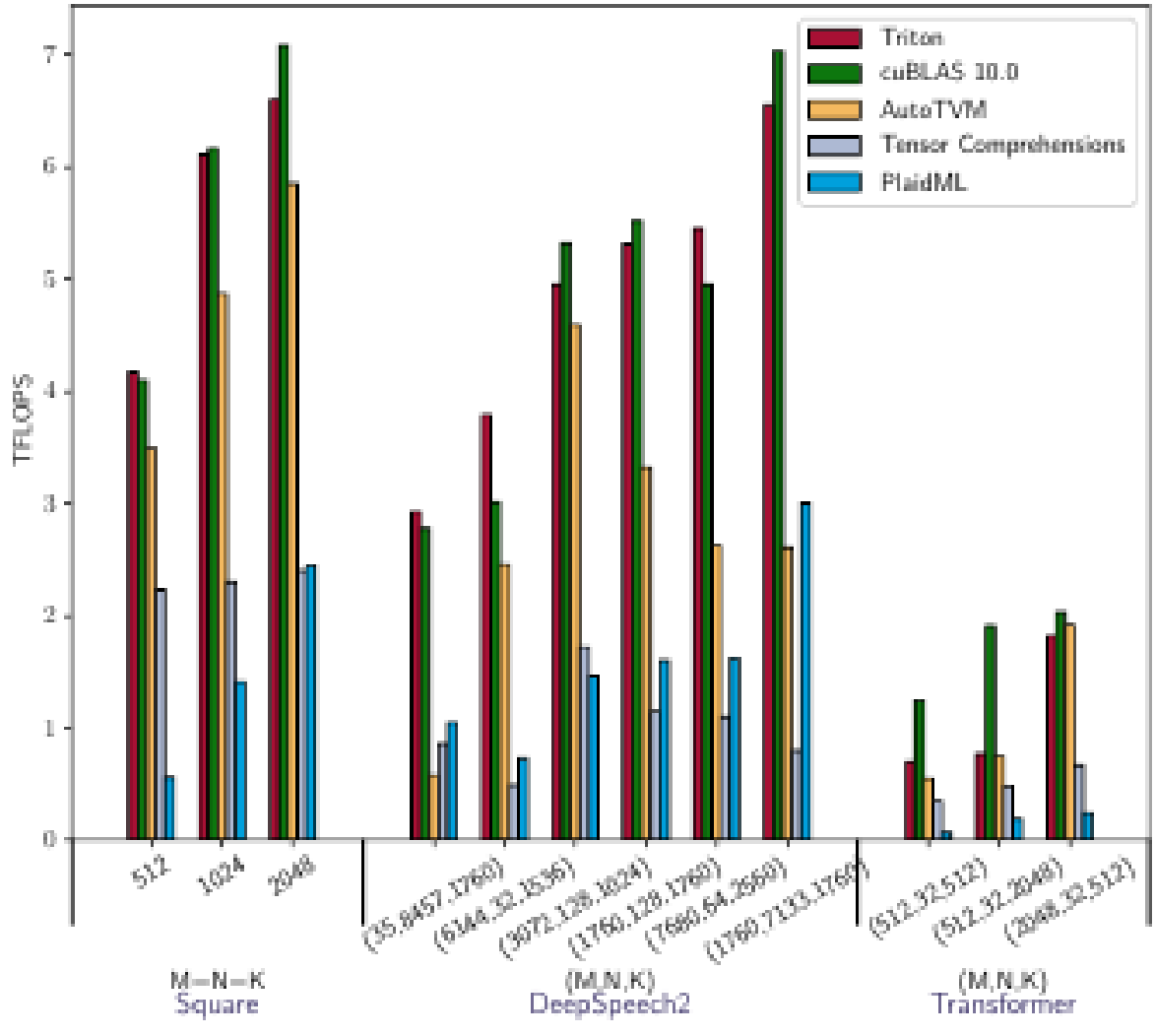
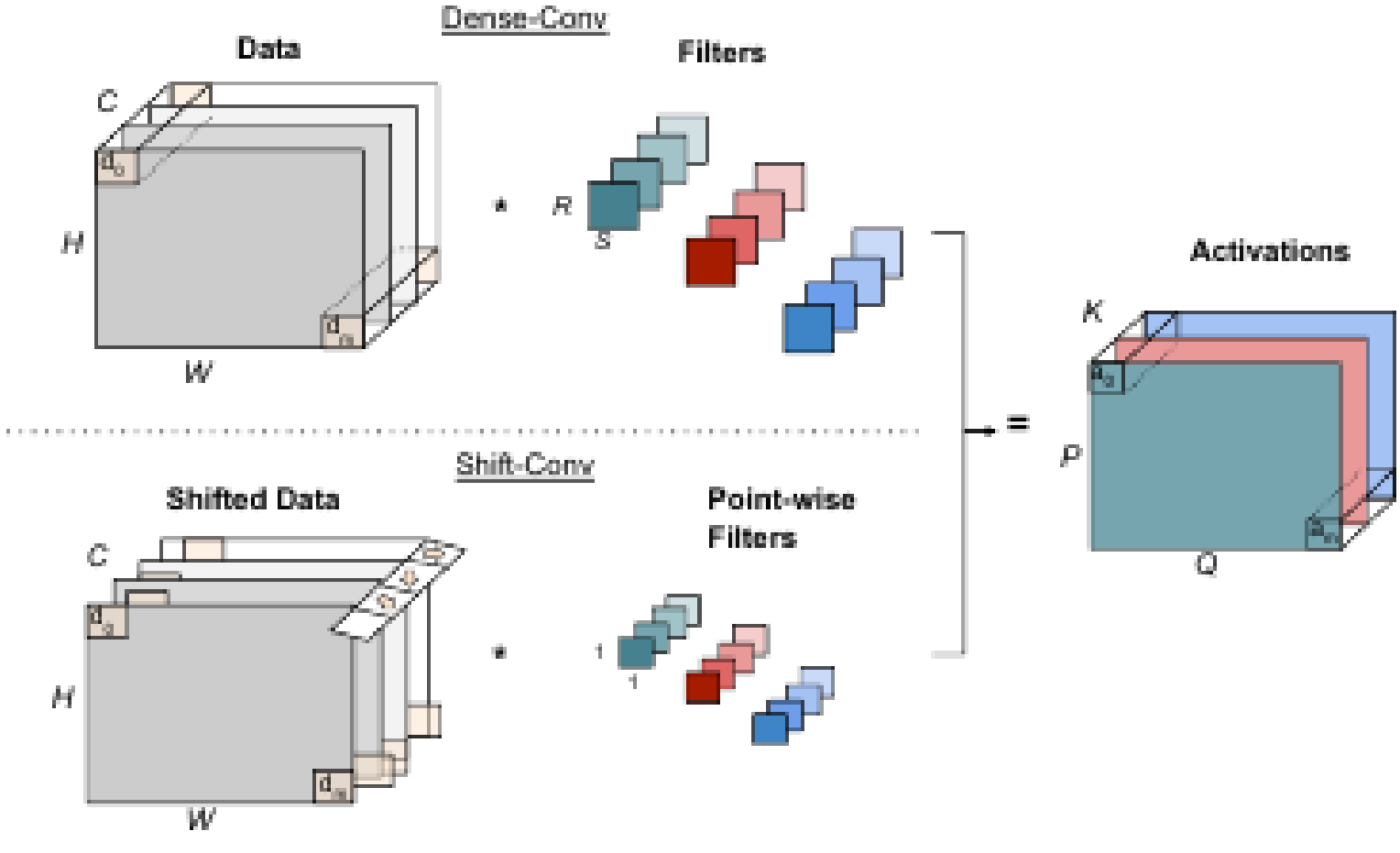


Рис.15. 8. Виконання множення матриці

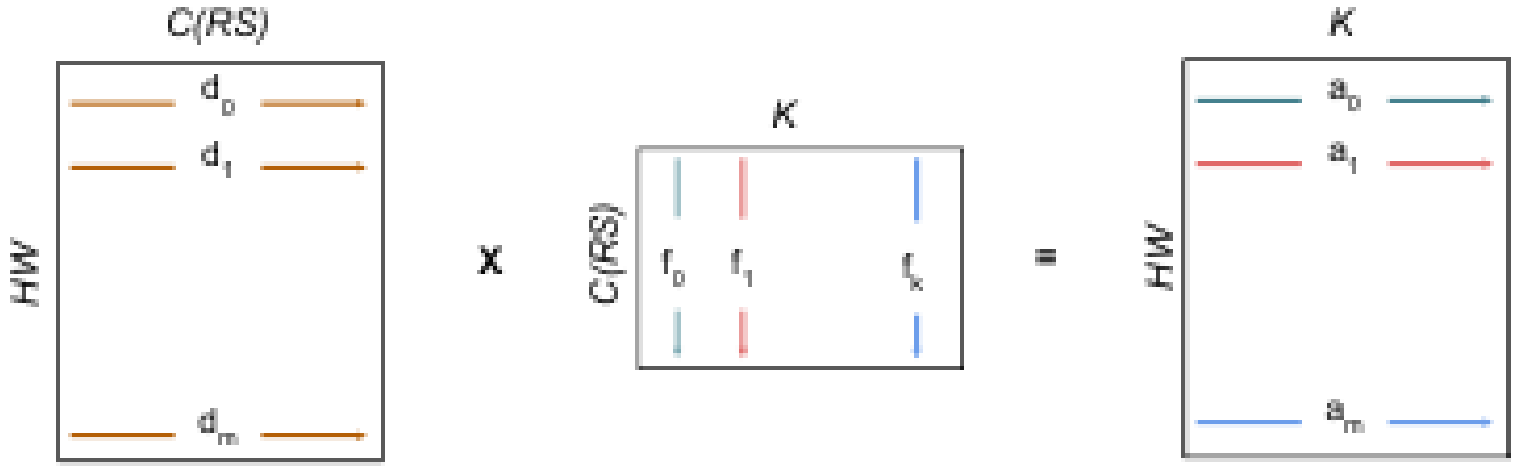
Triton і cuBLAS, як правило, нарівні один з одним і досягають більш ніж 90% максимальної продуктивності пристрою при виконанні певних завдань. CuBLAS, однак, залишається швидшим, ніж Triton на неглибоких трансформаторних нейронних мережах завдяки використанню 3D-алгоритму [2], який розбиває глибокі скорочення на незалежні фрагменти, щоб забезпечити більше паралельності, коли M і N занадто малі. В іншому випадку існуючі DSL працюють у 2-3 рази повільніше, ніж наше рішення, за винятком TVM (< 2 рази повільніше), коли вхідні форми кратні 32.

15.6.2 Згортки

Згорткові нейронні мережі (CNN) є важливим класом моделей машинного навчання, які повинні добре підтримуватися DSL та компіляторами. Вони засновані на згорткових шарах (Малюнок 15.9a), реалізація яких у вигляді множення матриці (Малюнок 15.9b) необхідна для використання спеціалізованого обладнання для обробки тензорів, але не підтримується існуючими DSL. Тут ми аналізуємо повторну реалізацію Triton алгоритму cuDNN «IMPLICIT_GEMM» (Розділ 15.6.2.1) і надаємо перше об'єднане ядро, доступне для зміщених згорток (Розділ 15.6.2.2). Ми реалізували ці підпрограми за допомогою пошукових таблиць приростів покажчика, як показано в лістінгу 15.8.



(a) A convolutional layer



(b) Equivalent matrix multiplication

Рис.15.9. Щільні та зміщені згорткові шари (a) у перегляді як матричне множення (b)

15.6.2.1 Щільні звивини

Згорткові шари, які розглядаються в цьому підрозділі, взяті з літератури з глибокого навчання та показані в таблиці 15.1.

Як показано на малюнку 15.10, Triton перевершує реалізацію IMPLICIT_GEMM cuDNN для ResNet. Це може бути пов’язано з тим, що cuDNN також підтримує кращі алгоритми для згорток 3 × 3 (тобто Winograd [25]), таким чином залишаючи мало інженерних ресурсів для оптимізації ядер меншого значення. Коли швидкі алгоритми недоступні (наприклад, DeepSpeech2), cuDNN і Triton на рівні.

Таблиця 1. Задачі на згортку, розглянуті в цій лекції

	H	W	C	B	K	R	S	Application
Task 1	112	112	64	4	128	3	3	ResNet [19]
Task 2	56	56	128	4	256	3	3	ResNet
Task 3	28	28	256	4	512	3	3	ResNet
Task 4	14	14	512	4	512	3	3	ResNet
Task 5	7	7	512	4	512	3	3	ResNet
Task 6	161	700	1	8	64	5	5	DeepSpeech2
Task 7	79	341	32	8	32	5	10	DeepSpeech2

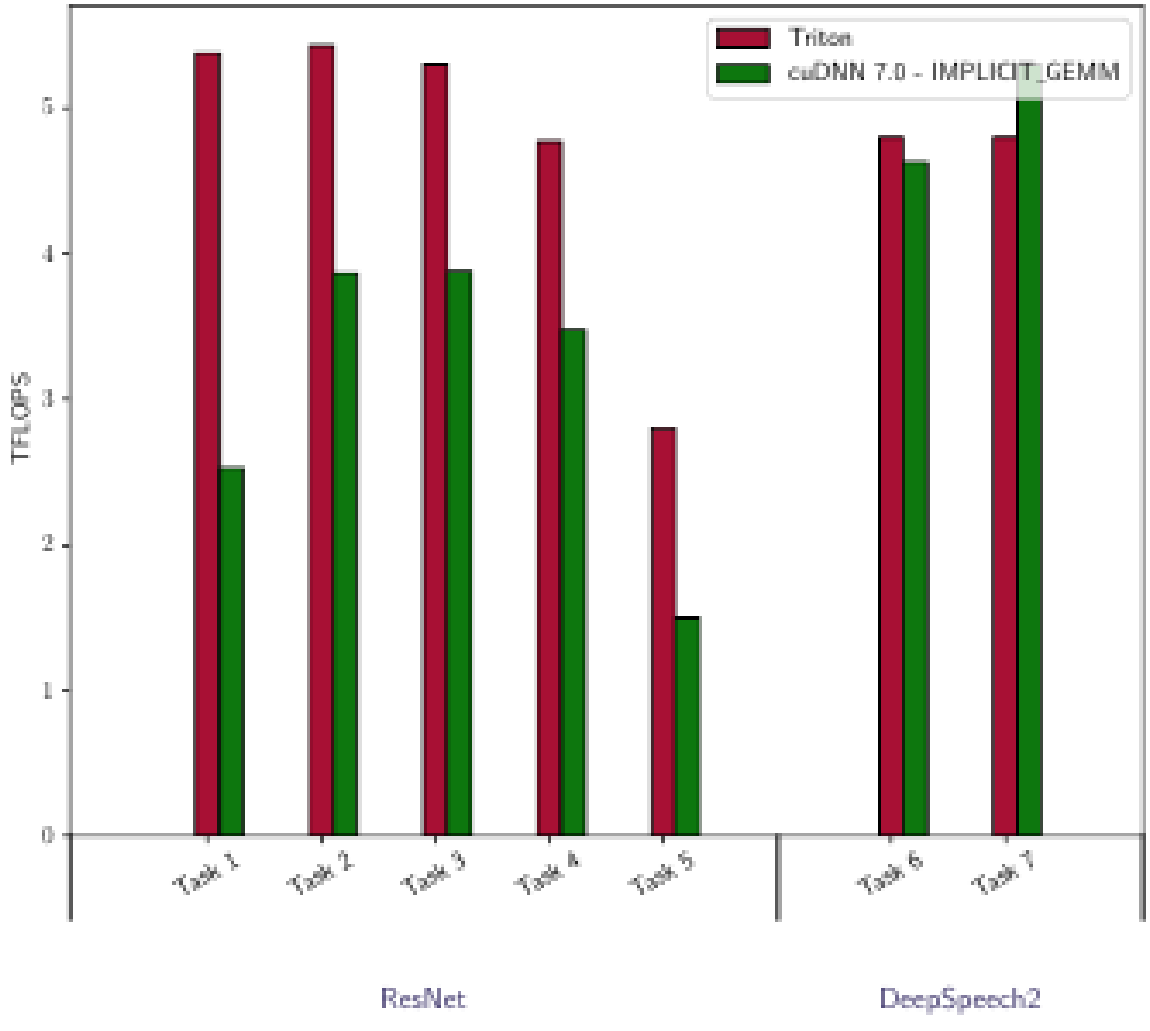


Рис.15.10. Виконання неявного множення матриці

15.6.2.2 Згортки зсуву

Нарешті ми розглядаємо реалізацію Завдання 1-5 з Таблиці 15.1 як зміщені згортки – новий підхід до CNN (див. Рисунок 15.9a). Ми порівнюємо нашу реалізацію об’єднаного модуля shift-conv у Triton (лістинг 15.8) з реалізацією наївної реалізації, що спирається на рукописне ядро зсуву та окремий виклик cuBLAS. Ми також повідомляємо про максимальну досяжну продуктивність, коли зсув не виконано (тобто згортка 1 × 1). Як ми бачимо на малюнку 15.11, наша реалізація Triton здатна майже повністю приховати вартість зміщення.

Лістінг 15.8. shift-згортки в Triton-C

```
const tunable int TM = {16 , 32 , 64 , 128};
const tunable int TN = {16 , 32 , 64 , 128};
const tunable int TK = {8};
__constant__ int * delta = alloc_const int
[512];
for ( int c = 0; c < C ; c ++)
delta [ c ] = c * H * W + shift_h [ c ]* W +
shift_w [ c ]
void shift_conv ( restrict read_only float
*a ,
restrict read_only float *b , float *c ,
int M , int N , int K ) {
int rxa [ TM ] = get_global_range [ TM ](0) ;
int ryb [ TN ] = get_global_range [ TN ](1) ;
```

```
int rka [ TK ] = 0 ... TK ;
int rkb [ TK ] = 0 ... TK ;
float C [ TM , TN ] = 0;
float * pxa [ TM , TK ] = a + rxa [: , newaxis
];
float * pb [ TN , TK ] = b + ryb [: ,
newaxis ] + rkb *N;
__constant__ int * pd [ TK ] = delta + rka ;
for ( int k = K ; k > 0; k = k - TK ) {
int delta [ TK ] = * pd ;
float * pa [ TM , TK ] = pxa + delta [ newaxis
, :];
float a [ TM , TK ] = * pa ;
float b [ TN , TK ] = * pb ;
C = dot (a , trans ( b ) , C ) ;
pb = pb + TK * N ;
```

```
pd = pd + TK ;
}
int rxc [ TM ] = get_global_range [ TM ](0) ;
int ryc [ TN ] = get_global_range [ TN ](1) ;
float * pc [ TM , TN ] = c + rxc [: ,
newaxis ] + ryc *M;
bool checkc0 [ TM ] = rxc < M ;
bool checkc1 [ TN ] = ryc < N ;
bool checkc [ TM , TN ] = checkc0 [: , newaxis
] && checkc1 ;
@checkc * pc = C ;
}
```

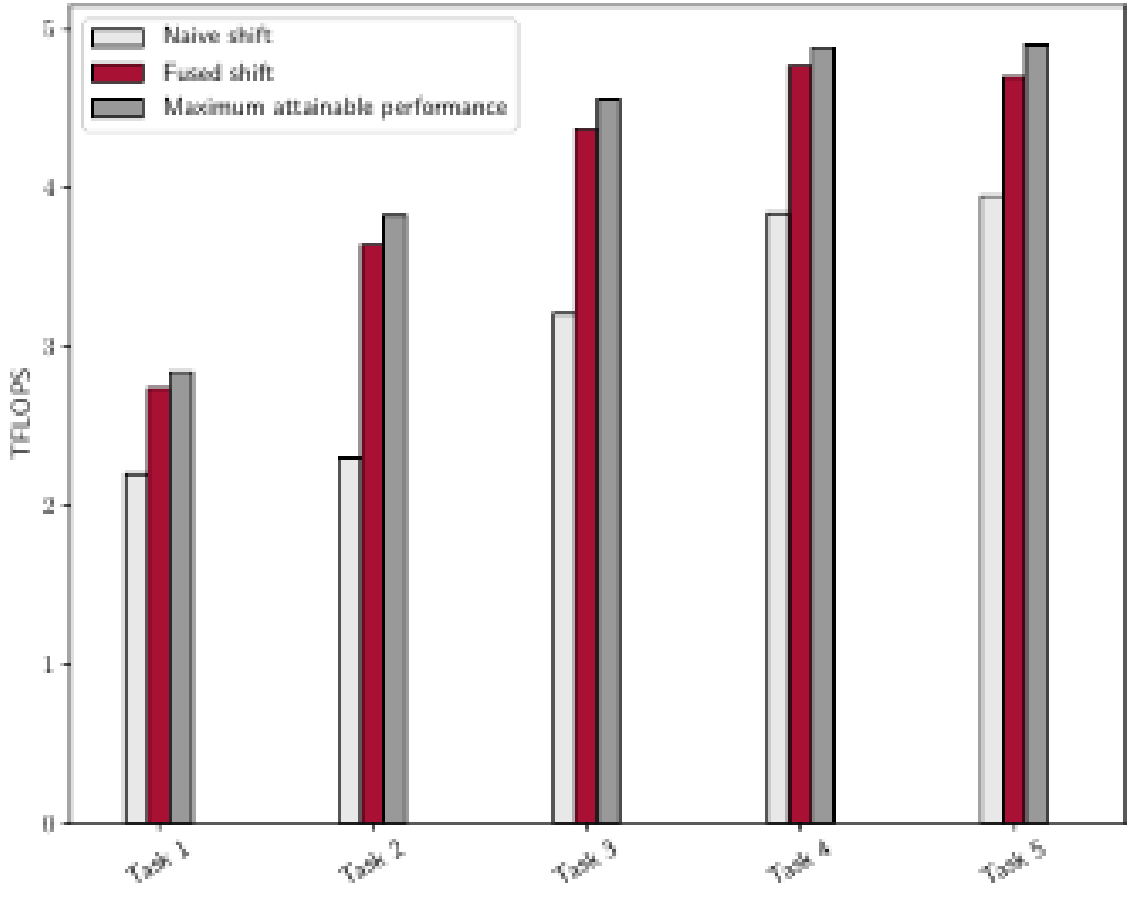


Рис.15.11. Виконання зміщених звивин у Тритоні

У цій лекції ми представили Triton, мову з відкритим вихідним кодом і компілятор для вираження та компіляції обчислень нейронної мережі з плитками в ефективний машинний код. Ми показали, що додавання лише кількох розширень потоку даних і керування до LLVM-IR може уможливити різні проходи оптимізації на рівні плитки, які разом призведуть до продуктивності на рівні з бібліотеками постачальників. Ми також запропонували Triton-C, мову вищого рівня, на якій ми змогли коротко реалізувати ефективні ядра для нових архітектур нейронних мереж для CNN.

Використана література

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-scale Machine Learning. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16).

[2] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A three-dimensional approach to parallel matrix multiplication. IBM Journal of Research and Development 39, 5 (Sep. 1995), 575–582. https://doi.org/10.1147/rd.395.0575

[3] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley,

Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. CoRR abs/1512.02595 (2015). arXiv:1512.02595 http://arxiv.org/abs/1512.02595

[4] M. Auguin and F. Larbey. [n. d.]. Opsila: an advanced SIMD for numerical analysis and signal processing. ([n. d.]), 311–318.

[5] Bin Bao and Chen Ding. 2013. Defensive Loop Tiling for Shared Cache. In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO) (CGO '13). IEEE Computer Society, Washington, DC, USA, 1–11.

[6] Muthu Manikandan Baskaran, Albert Hartono, Sanket Tavarageri, Thomas Henretty, J. Ramanujam, and P. Sadayappan. 2010. Parameterized Tiling Revisited. In Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '10). ACM, New York, NY, USA, 200–209. https://doi.org/10.1145/1772954.1772983

[7] Matthias Braun, Sebastian Buchwald, Sebastian Hack, Roland Leissa,

Christoph Mallon, and Andreas Zwinkau. 2013. Simple and Efficient Construction of Static Single Assignment Form. In Proceedings of the 22Nd International Conference on Compiler Construction (CC'13). 102–122.
[8] Lori Carter, Beth Simon, Brad Calder, Larry Carter, and Ferrante. [n. d.]. Predicated Static Single Assignment. In Proceedings of the PACT 1999 Conference on Parallel Architectures and Compilation Techniques.
[9] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. CoRR abs/1512.01274 (2015).
[10] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. CoRR abs/1802.04799 (2018).
[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q. Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: End-to-End Optimization Stack for Deep Learning. CoRR abs/1802.04799 (2018).

[12] Jack W. Davidson and Sanjay Jinturkar. 1994. Memory Access Coalescing: A Technique for Eliminating Redundant Memory Accesses. In Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI '94). ACM, New York, NY, USA, 186–195. https://doi.org/10.1145/178243.178259
[13] Greg Diamos, Shubho Sengupta, Bryan Catanzaro, Mike Chrzanowski, Adam Coates, Erich Elsen, Jesse Engel, Awni Hannun, and Sanjeev Satheesh. 2016. Persistent RNNs: Stashing Recurrent Weights On-Chip. In Proceedings of The 33rd International Conference on Machine Learning.
[14] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. 2018. Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs. In Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2018). ACM, New York, NY, USA, 42–51. https://doi.org/10.1145/3211346.3211354
[15] Jordan Gergov. 1999. Algorithms for Compile-time Memory Optimization. In Proceedings of the Tenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '99).

[16] Google Inc. 2017. Tensorflow XLA. https://www.tensorflow.org/performance/xla/
[17] Scott Gray and Alex Radford ans Diederik P. Kingma. 2017. Gpu kernels for block-sparse weights. CoRR abs/1711.09224 (2017).
[18] M. Griebl, C. Lengauer, and S. Wetzel. 1998. Code generation in the polytope model. In Proceedings. 1998 International Conference on Parallel Architectures and Compilation Techniques (Cat. No.98EX192). 106–111. https://doi.org/10.1109/PACT.1998.727179
[19] K. He, X. Zhang, S. Ren, and J. Sun. 2016. Deep Residual Learning for Image Recognition. In 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 770–778. https://doi.org/10.1109/CVPR.2016.90
[20] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. CoRR abs/1704.04861 (2017). arXiv:1704.04861 http://arxiv.org/abs/1704.04861
[21] Jianyu Huang and Robert A. van de Geijn. 2016. BLISlab: A Sandbox for Optimizing GEMM. FLAME Working Note #80, TR-16-13. The

University of Texas at Austin, Department of Computer Science. http://arxiv.org/pdf/1609.00076v1.pdf
[22] Intel AI. 2018. PlaidML. https://www.intel.ai/reintroducing-plaidml
[23] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. [n. d.]. CUTLASS: Fast Linear Algebra in CUDA C++. ([n. d.]). https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/
[24] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'12).
[25] Andrew Lavin and Scott Gray. 2016. Fast Algorithms for Convolutional Neural Networks. (CVPR'16).
[26] Darryl D. Lin, Sachin S. Talathi, and V. Sreekanth Annapureddy. 2016. Fixed Point Quantization of Deep Convolutional Networks. In Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48 (ICML'16). JMLR.org, 2849–2858. http://dl.acm.org/citation.cfm?id=3045390.3045690
[27] Matt Martineau, Patrick Atkinson, and Simon McIntosh-Smith. 2017. Benchmarking the NVIDIA V100 GPU and Tensor Cores. Springer.

[28] Bradley McDanel, Surat Teerapittayanon, and H.T. Kung. 2017. Embedded Binarized Neural Networks. In Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks (EWSN ’17). Junction Publishing, USA, 168–173. http://dl.acm.org/citation.cfm?id=3108009.3108031
[29] W. M. McKeeman. 1965. Peephole Optimization. Commun. ACM 8, 7 (July 1965), 443–444. https://doi.org/10.1145/364995.365000
[30] Sanyam Mehta, Rajat Garg, Nishad Trivedi, and Pen-Chung Yew. 2016. TurboTiling: Leveraging Prefetching to Boost Performance of Tiled Codes. In Proceedings of the 2016 International Conference on Supercomputing (ICS ’16). ACM, New York, NY, USA, Article 38, 12 pages.
[31] Sharan Narang, Eric Undersander, and Gregory F. Diamos. 2017. Block-Sparse Recurrent Neural Networks. CoRR abs/1711.02782 (2017).
[32] Rajib Nath, Stanimire Tomov, and Jack Dongarra. 2011. Accelerating GPU Kernels for Dense Linear Algebra. In Proceedings of the 9th International Conference on High Performance Computing for Computational Science (VECPAR’10). Springer-Verlag, Berlin, Heidelberg, 83–92. http://dl.acm.org/citation.cfm?id=1964238.1964250
[33] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. 2008.

Scalable Parallel Programming with CUDA. Queue 6, 2 (March 2008), 40–53. https://doi.org/10.1145/1365490.1365500
[34] Cedric Nugteren. 2017. CLBlast: A Tuned OpenCL BLAS Library. CoRR abs/1705.05249 (2017). arXiv:1705.05249 http://arxiv.org/abs/1705.05249
[35] Travis Oliphant. 2006–. NumPy: A guide to NumPy. USA: Trelgol Publishing. (2006–). http://www.numpy.org/ [Online; accessed <today>].
[36] PyTorch. 2016. . https://github.com/pytorch/pytorch
[37] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13).
[38] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. CoRR abs/1805.00907 (2018).

[39] Arthur Stoutchinin and Francois de Ferriere. 2001. Efficient Static Single Assignment Form for Predication. In Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34). IEEE Computer Society, Washington, DC, USA, 172–181. http://dl.acm.org/citation.cfm?id=563998.564022
[40] Philippe Tillet and David Cox. 2017. Input-aware Auto-tuning of Compute-bound HPC Kernels. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17). ACM.
[41] Didem Unat, Tan Nguyen, Weiqun Zhang, Muhammed Nufail Farooqi, Burak Bastem, George Michelogiannakis, Ann Almgren, editor="Kunkel Julian M. Shalf, John", Pavan Balaji, and Jack Dongarra. 2016". TiDA: High-Level Programming Abstractions for Data Locality Management.
[42] Field G. Van Zee and Robert A. van de Geijn. 2015. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. ACM Trans. Math. Softw. 41, 3 (June 2015).
[43] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew

Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. CoRR abs/1802.04730 (2018).
[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. CoRR abs/1706.03762 (2017). arXiv:1706.03762 http://arxiv.org/abs/1706.03762
[45] R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. 2000. Automated Empirical Optimization of Software and the ATLAS Project. PARALLEL COMPUTING 27 (2000), 2001.
[46] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. Commun. ACM 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785
[47] Bichen Wu, Alvin Wan, Xiangyu Yue, Peter H. Jin, Sicheng Zhao, Noah Golmant, Amir Gholaminejad, Joseph Gonzalez, and Kurt Keutzer. 2017. Shift: A Zero FLOP, Zero Parameter Alternative to Spatial Convolutions. CoRR abs/1711.08141 (2017).
[48] Jiacheng Zhao, Huimin Cui, Yalin Zhang, Jingling Xue, and Xiaobing

Feng. 2018. Revisiting Loop Tiling for Datacenters: Live and Let Live. In Proceedings of the 2018 International Conference on Supercomputing (ICS ’18). ACM, New York, NY, USA, 328–340.

Наступна лекція буде присвячена парадигмі мультиагентного програмування.