DOI: 10.15514/ISPRAS-2025-37(1)-5

Подход к построению компиляторов нейронных сетей с использованием инфраструктуры MLIR

[1,2] *И.И. Кулагин, ORCID: 0000-0003-2191-1578 <i.kulagin@ispras.ru>*
[1] *Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1] *М.В. Пантилимонов, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>*
[1,3] *А.В. Вязовцев, ORCID: 0009-0007-0826-2186 <andrey.vyazovtsev@ispras.ru>*
[1,2] *М.М. Романов, ORCID: 0009-0001-3242-1171 <mmromanov@ispras.ru>*
[1,2] *Д.М. Мельник, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, 109004, Россия, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет имени М.В. Ломоносова, 119991, Россия, Москва, Ленинские горы, д. 1.*
[3] *Московский физико-технический институт, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9*

Аннотация. Развитие матричных расширений процессорных архитектур, а также внедрение этих расширений в специализированные AI-процессоры, позволяет существенно повысить эффективность выполнения искусственных нейронных сетей. В работе выполнен обзор базовых функциональных возможностей некоторых популярных матричных расширений процессорных архитектур, в частности расширений ARM SME, RISC-V IME, RISC-V AME, а также процессорной архитектуры DaVinci. В результате проведенного анализа была предложена модель абстрактного матричного процессора, отражающая особенности современных процессорных архитектур, которые поддерживают матричное расширение. Для введенной модели матричного процессора разработано гетерогенное матричное промежуточное представление, которое может быть использовано для построения компиляторов нейронных сетей. Предложенное промежуточное представление было реализовано в инфраструктуре MLIR в виде диалекта heteroMx. В работе также описан подход к построению AI-компилятора с использованием разработанного диалекта heteroMx. Разработанное промежуточное представление может быть адаптировано или конкретизировано для других матричных процессорных архитектур.

Ключевые слова: матричное расширение; архитектуры RISC-V; расширенная архитектура матричных вычислений ARM SME; промежуточное представление; инфраструктура MLIR.

Для цитирования: Кулагин И.И., Бучацкий Р.А., Пантилимонов М.В., Вязовцев А.В., Романов М.М., Мельник Д.М. Подход к построению компиляторов нейронных сетей с использованием инфраструктуры MLIR. Труды ИСП РАН, том 37, вып. 1, 2025 г., стр. 87–106. DOI: 10.15514/ISPRAS–2025–37(1)–5.

Approach to Building AI-Compilers Using the MLIR Framework

[1,2] *I.I. Kulagin, ORCID: 0000-0003-2191-1578 <i.kulagin@ispras.ru>*
[1] *R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1] *M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>*
[1,3] *A.V. Vyazovtsev, ORCID: 0009-0007-0826-2186 <andrey.vyazovtsev@ispras.ru>*
[1,2] *M. M. Romanov, ORCID: 0009-0001-3242-1171 <mmromanov@ispras.ru>*
[1,2] *D.M. Melnik, ORCID: 0000-0002-0177-1558 <dm@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] *Moscow Institute of Physics and Technology,*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.

Abstract. The development of matrix extensions of processor architectures, as well as the implementation of these extensions in specialized AI processors, can significantly improve the efficiency of artificial neural networks. The paper provides an overview of the basic functionality of some popular matrix extensions of processor architectures, in particular, ARM SME, RISC-V IME, RISC-V AME extensions, as well as the DaVinci processor architecture. As a result of the analysis, a model of an abstract matrix processor was proposed. This model reflects the features of modern processor architectures supporting matrix extensions. For the introduced model of the matrix processor, a heterogeneous matrix intermediate representation was developed, which can be used to build compilers for neural networks. The proposed intermediate representation was implemented in the MLIR infrastructure as a heteroMx dialect. The paper also describes an approach to building an AI compiler using the heteroMx dialect. The developed intermediate representation can be adapted or specified for other matrix processor architectures.

Keywords: matrix extension; RISC-V; ARM SME; AI-compiler intermediate representation; MLIR infrastructure.

For citation: Kulagin I.I., Buchatskiy R.A., Pantilimonov M.V., Vyazovtsev A.V., Romanov M. M., Melnik D.M. Approach to Building AI-Compilers Using the MLIR Framework. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 1, 2025, pp. 87-106 (in Russian). DOI: 10.15514/ISPRAS-2025-37(1)-5.

1. Введение

Значительная часть вычислений при выполнении искусственных нейронных сетей приходится на операции свертки и умножения матриц. Для ускорения этих операций в частности и всей нейронной сети вообще активно ведется разработка расширений процессорных архитектур, которые реализуют матричные инструкции. Среди наиболее известных матричных расширений можно выделить расширение Scalable Matrix Extension архитектуры ARM, расширения Integrated Matrix Extension и Attached Matrix Extension архитектуры RISC-V. Также можно отметить процессорную архитектуру DaVinci от компании Huawei, нейроморфный процессор NeuroMorphic Processor (NMP) [1] от компании LG и тензорный процессор от компании Google (Google TPU) [2] и др.

Матричное расширение процессорной архитектуры обычно содержит множество матричных регистров и специальные инструкции, которые реализуют умножение матриц. Матричное расширение может быть разработано на основе векторного расширения. В этом случае матричные регистры реализуются на основе регистрового файла векторного расширения, например, путем объединения части векторных регистров в матрицу или использования векторного регистра для хранения небольшого блока матрицы. Такое матричное расширение называется интегрированным. В случае, если матричное расширение не использует векторный регистровый файл, оно называется независимым. Процессоры, реализующие

интегрированное расширение, отличаются значительно меньшей стоимостью по сравнению с теми, которые реализуют независимое расширение.

Генерация эффективного кода для матричных процессорных архитектур проблематична в условиях отсутствия высокоуровневого промежуточного представления (Intermediate Representation – IR). Используемый в качестве промышленного стандарта LLVM IR не подходит в качестве такого IR, так как является слишком низкоуровневым. По этой причине в работе предложено гетерогенное матричное промежуточное представление.

Реализация предложенного промежуточного представления требует разработки объемной инфраструктуры, включающей в себя менеджер проходов, движок сопоставления с образцом, интерфейсы для трансляции инструкции одних промежуточных представлений в инструкции других и т.д. Одним из перспективных подходов к разработке промежуточных представлений является использование инфраструктуры проекта MLIR. Инфраструктура MLIR позволяет автоматизировать процесс разработки промежуточных представлений различных уровней. Предложенное гетерогенное матричное IR реализовано с использованием инфраструктуры MLIR [3].

Дальнейшее повествование в работе ведется по следующему плану. Во втором разделе будет выполнен обзор базовых возможностей существующих матричных расширений процессорных архитектур. После чего будет введена модель абстрактного матричного процессора. В третьем разделе будут раскрыты детали предложенного матричного промежуточного представления, выполнен короткий обзор инфраструктуры MLIR. Далее будет представлена функциональная структура AI-компилятора. В четвертом разделе содержатся выводы и обозначены некоторые перспективные направления для дальнейших исследований.

2. Модель архитектуры процессора с матричным расширением

В данном разделе представлена модель абстрактной архитектуры процессора, поддерживающая матричное расширение набора команд. Предлагаемая модель отражает некоторые аспекты существующих матричных расширений и процессорных архитектур таких, как матричное расширение ARM Scalable Matrix Extension (ARM SME), RISC-V Integrated Matrix Extension (RISC-V IME), RISC-V Attached Matrix Extension (RISC-V AME), Huawei Da Vinci. К числу таких аспектов можно отнести особенности реализации иерархической организации памяти, семантику инструкций матричного расширения и режимы работы с матричными регистрами.

2.1 Расширение ARM Scalable Matrix Extension

Расширение ARM Scalable Matrix Extension было включено в архитектуру ARM для A-профиля 20 марта 2024 года. SME [4] является продолжением развития масштабируемого векторного расширения Scalable Vector Extension (SVE) [5]. SVE добавляет в архитектурное состояние следующие компоненты: 32 масштабируемых векторных регистра (Z0-Z31), 16 предикатных регистров (P0-P15), специальный регистр first-fault register (FFR), управляющие регистры для различных режимов работы процессора (ZCR_EL1-ZCR_EL3). Масштабируемые векторные регистры не имеют фиксированной длины, каждая реализация процессора может содержать векторные регистры необходимой длины. Архитектура только накладывает ограничение по кратности на длину регистров. Она должна быть кратна 128 битам в диапазоне от 128 до 2048 бит. Масштабируемые регистры SVE расширения позволяют абстрагироваться от конкретной длины векторных регистров при разработке векторизованных программ, а компилятор генерирует код, содержащий универсальные инструкции, не зависящие от конкретной реализации архитектуры. Также SVE содержит множество инструкций: инструкции загрузки данных в масштабируемые регистры и

выгрузки из них в память, инструкции арифметических и логических операций над элементами векторных регистров, инструкции для спекулятивной векторизации и др.

Позже расширение SVE было расширено дополнительными инструкциями до SVE2. Были добавлены инструкции построения гистограмм, инструкции сопоставления с образцом, инструкции, реализующие криптографические алгоритмы, и др.

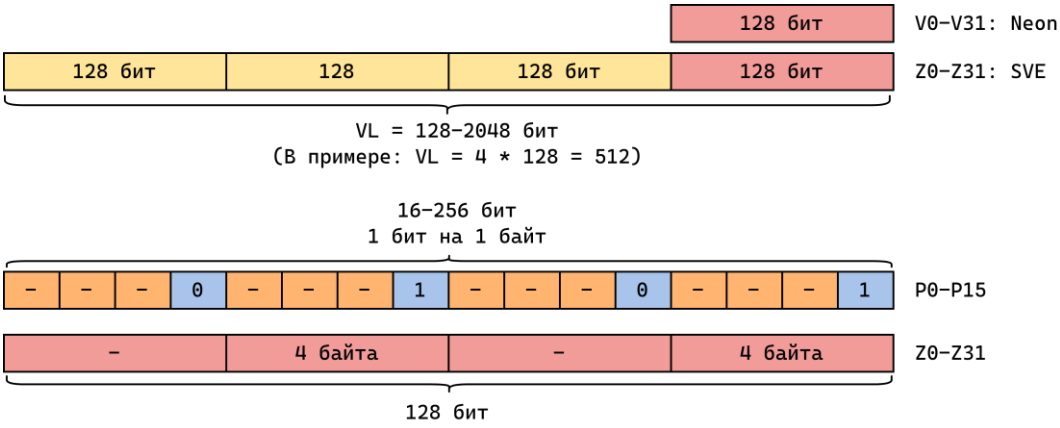


Рис. 1. Структура масштабируемых векторных регистров ARM SVE.
Fig. 1. The structure of ARM SVE scalable vector registers.

На рис. 1 представлена структура масштабируемых векторных регистров, а также масштабируемых предикатных регистров. Длина вектора неизвестна на этапе компиляции и может отличаться на различных реализациях архитектуры. В данном примере длина вектора равна 512 бит, т.е. 4 × 128 бит, что удовлетворяет ограничениям SVE, а масштаб векторного регистра (V scale) равен 4. Набор команд расширения SVE спроектирован таким образом, чтобы избежать в коде использования конкретной длины векторных регистров. Для этой цели расширение содержит 16 предикатных регистров. Предикатные регистры выступают в роли флага использования векторной инструкцией соответствующего элемента векторного регистра. Каждому биту предикатного регистра соответствует 8 бит векторного регистра. Таким образом, длина предикатных регистров в зависимости от реализации архитектуры может варьироваться от 16 до 256 бит.

На рис. 2 представлен пример вычисления SAXPY с использованием SVE инструкций. В этом примере особый интерес представляет инструкция whilelt. Эта инструкция инициализирует предикатный регистр, исходя из результата сравнения i < n, количества оставшихся для выполнения итераций с фактической длиной масштабируемых векторных регистров.

Расширение SME включает в себя SVE2 и определяет следующие ключевые возможности:

- масштабируемый двумерный ZA регистр (матричный регистр), позволяющий хранить двумерные матричные блоки;
- режим Streaming SVE (SSVE) – потоковый режим работы с векторными инструкциями расширения SVE;
- инструкции, вычисляющие внешнее произведение векторов в результирующий матричный регистр, используемый как аккумулятор;

инструкции загрузки, хранения и перемещения, которые передают вектор в строку или столбец матричного блока.

Регистр ZA используется матричными инструкциями как аккумулятор. Этот регистр состоит из виртуальных матричных блоков (tiles), количество и размер которых определяются

размером типа элементов. В табл. 1 представлены возможные конфигурации матричного регистра для различных типов элементов.

```c
void saxpy(float *x, float *y, float a, int n) {
  for (int i = 0; i < n; i++)
    y[i] = a * x[i] + y[i];
}
```

а) Реализация saxpy на языке C
a) Implementation of saxpy in C

```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy:
  ldrsw x3, [x3]            // x3=*n
  mov x4, #0                // x4=i=0
  ldr s0, [x2]              // s0=*a
  b .latch
.loop:
  ldr s1, [x0, x4, lsl #2]  // s1=x[i]
  ldr s2, [x1, x4, lsl #2]  // s2=y[i]
  fmadd s2, s1, s0, s2      // s2+=x[i]*a
  str s2, [x1, x4, lsl #2]  // y[i]=s2
  add x4, x4, #1            // i+=1
.latch:
  cmp x4, x3                // i < n
  b.lt .loop                // more to do?
  ret
```

б) Скалярная реализация saxpy на ассемблере
b) Scalar implementation of saxpy in assembly

```
// x0 = &x[0], x1 = &y[0], x2 = &a, x3 = &n
saxpy:
  ldrsw x3, [x3]          // x3=*n
  mov x4, #0              // x4=i=0
  whilelt p0.w, x4, x3    // p0=while(i++<n)
  ld1rw z0.w, p0/z, [x2]  // p0:z0=bcast(*a)
.loop:
  ld1w z1.w, p0/z, [x0, x4, lsl #3] // p0:z1=x[i]
  ld1w z2.w, p0/z, [x1, x4, lsl #3] // p0:z2=y[i]
  fmla z2.w, p0/m, z1.w, z0.w       // p0?z2+=x[i]*a
  st1w z2.w, p0, [x1, x4, lsl #3]   // p0?y[i]=z2
  incw x4                          // i+=(VL/32)
.latch:
  whilelt p0.w, x4, x3  // p0=while(i++<n)
  b.first .loop         // more to do?
  ret
```

в) Реализация saxpy
с использованием ARM SVE инструкций
c) Implementation of saxpy using ARM SVE instructions

Рис. 2. Пример реализации saxpy на языке C и ассемблере с использованием инструкций ARM SVE.
Fig. 2. Implementation of saxpy in C and assembly using ARM SVE instructions.

Табл. 1. Зависимость размера виртуальных блоков регистра ZA от типа элемента.
Table 1. Dependency of the size of the virtual blocks of the register ZA on the element type.

Тип элемента	Кол-во блоков (tiles)	Размер виртуального блока	Название виртуальных блоков	Размер регистра Z0-Z31
i8	1	(16 * vscale) × (16 * vscale)	ZA0.B	16 * vscale
i16	2	(8 * vscale) × (8 * vscale)	ZA0-ZA1.H	8 * vscale
i32/f32	4	(4 * vscale) × (4 * vscale)	ZA0-ZA3.S	4 * vscale
i64/f64	8	(2 * vscale) × (2 * vscale)	ZA0-ZA7.D	2 * vscale
i128	16	(1 * vscale) × (1 * vscale)	ZA0-ZA15.Q	1 * vscale

Физический размер матричного регистра ZA зависит от конкретной реализации архитектуры, как и в случае с масштабируемыми векторными регистрами. Спецификация SME лишь накладывает ограничения на размер матричного регистра. Его размер определяется длиной потокового вектора (Streaming Vector Length – SVL), то есть длиной масштабируемых регистров Z0-Z31 в потоковом режиме, и должен быть равен SVL × SVL. Таким образом, SME-код не оперирует конкретными размерами матричных блоков, так как их размер является неизвестным и может отличаться на различных реализациях архитектуры. Например, на процессоре Apple M4 длина потокового вектора SVL равна 512 бит, следовательно, данная реализация содержит 32 регистра Z0-Z31 длиной 512 / 8 = 64 байт, а также матричный регистр ZA размером 64 × 64 = 4096 байт. На рис. 3 представлен пример организации ZA регистра для случая, когда размер типа элемента равен 32 бита.

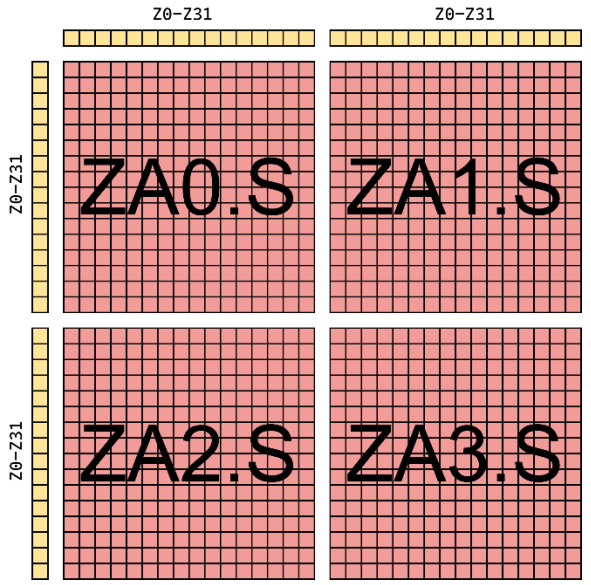


Рис. 3. Организация матричного регистра ZA. SVE = 512; типа элемента i32.
Fig. 3. Organization of the matrix register ZA with SVE = 512 and element type i32.

Ключевыми инструкциями SME являются инструкции внешнего произведения. Эти инструкции принимают в качестве входных операндов два масштабируемых SVE-регистра, хранящих вектора, и накапливают в матричном регистре ZA результат. Мнемоники инструкций внешнего произведения заканчиваются на MOPA. Например, инструкция внешнего произведения для чисел с плавающей запятой – FMOPA. Для типа FP32 инструкция FMOPA вычисляет внешнее произведение двух векторов размером SVL/32 элементов, расположенных в двух регистрах Z, и аккумулирует результат в регистре ZA размером SVL/32 × SVL/32 элементов. Поскольку у M4 SVL равна 512 битам, это означает, что инструкция FP32 FMOPA вычисляет внешнее произведение двух векторов по 16 элементов каждый и аккумулирует результат в регистре ZA размером 16 × 16 элементов. Таким образом, одна инструкция FP32 FMOPA на M4 выполняет 16 × 16 × 2 = 512 операций с плавающей запятой. Для выполнения инструкций SME необходимо переключить процессорное ядро в режим Streaming SVE. Активация этого режима осуществляется при помощи инструкции **SMSTART**, после чего архитектурное состояние SME становится доступным. Аналогично инструкция **SMSTOP** отключает режим SSVE. При переключении режимов происходит очистка регистров ZA и Z.

На рис. 4 приведен фрагмент SME-микроядра, выполняющего умножение матриц с элементами типа FP32 для одного блока (C += ABT). Матрицы A и C хранятся по столбцам, а матрица B по строкам. Фрагмент кода содержит только внутренний цикл, результат вычислений накапливается в регистре ZA. Так как для элементов FP32 регистр ZA состоит из 4 виртуальных блоков, то в результате вычисления данного цикла будет получен один блок матрицы C размером до (4 × SVL/32 × SVL/32) элементов. В случае выполнения данного фрагмента кода на процессоре Apple M4 блок матрицы C может быть размером до 32 × 32 элементов.

В теле цикла сперва загружается фрагмент столбца матрицы A (строка 5). Для процессора Apple M4 размер загруженного фрагмента может быть до 32 элементов. В случае если размер столбца меньше или не кратен 32 элементам, предикатный регистр PN8 обеспечит загрузку максимально возможного числа элементов. Далее, таким же образом загружаются элементы строки матрицы B (строка 6). Две инструкции в строках 7 и 8 инкрементируют адреса в X0 и

X1, указывая на следующий столбец A и следующую строку B. Инструкции FMOPA в строках 9-12 вычисляют четыре соответствующих внешних произведения и обновляют виртуальные матричные блоки ZA. После завершения цикла K (строки 3, 4 и 13) заканчивается вычисление блока C, и результат вычислений записывается из матричного регистра ZA в память. Далее вычисления будут повторяться для следующих блоков матрицы C.

```
1  // set predicate registers
2  // set register offset
3  k_loop:
4    sub x8, x8, #0x1
5    ld1w {z0.s, z1.s}, pn8/z, [x0]
6    ld1w {z2.s, z3.s}, pn9/z, [x1]
7    add x0, x0, x9
8    add x1, x1, x10
9    fmopa za0.s, p1/m, p0/m, z2.s, z0.s
10   fmopa za1.s, p1/m, p2/m, z2.s, z1.s
11   fmopa za2.s, p3/m, p0/m, z3.s, z0.s
12   fmopa za3.s, p3/m, p2/m, z3.s, z1.s
13 cbnz x8, k_loop
```

Рис. 4. Фрагмент микроядра, вычисляющий умножение матриц с использованием ARM SME инструкций.
Fig. 4. Fragment of the microkernel performing matrix multiplication using ARM SVE instructions.

2.2 Интегрированное матричное расширение RISC-V IME

Интегрированное матричное расширение RISC-V IME разрабатывается компанией SpacemiT и по состоянию на конец 2024 года находится в стадии разработки [6]. Это расширение основано на стандартных векторных регистрах векторного расширения RISC-V «V» (RVV) и не добавляет новых регистров управления состоянием и матричных регистров. Матричные регистры образуются путем группировки нескольких векторных регистров в единый блок. Модель программирования с использованием матричных инструкций остается максимально похожей на модель программирования с использованием векторного расширения RVV. Например, одинаковое использование режимов округления, инструкций конфигурации таких, как vsetvli/vsetivli/vsetvl и т. д. В отличие от векторного расширения RVV IME-инструкции выбирают соответствующий матричный блок для умножения матриц и аккумулирования результата на основе значений параметра длины вектора (Vector Length – VL) и размера элемента вектора (Single Element Width – SEW), установленных инструкцией конфигурирования vsetvli/vsetivli/vsetvl. Далее следует небольшой обзор векторного расширения RVV, а после – интегрированного матричного расширения RISC-V IME.

Расширение RVV идейно схоже с расширением ARM SVE. Оно содержит 32 масштабируемых векторных регистра v0-v31, набор векторных инструкций, системные регистры и специальные инструкции для управления динамическим состоянием. Каждый векторный регистр V имеет длину VLEN, неизвестную на этапе компиляции или написании кода и отличающуюся на различных реализациях расширения RVV. Регистры могут быть сгруппированы. Группа регистров может быть использована как единый вектор соответствующего размера. Размер группы может быть динамически изменен и определяется параметром LMUL. Размер элемента в векторе SEW также может быть динамически изменен.

Динамическое состояние процессора, определяющее поведение векторных инструкций, контролируется системными регистрами vector type register(vtype), vector length register(vl), vector byte length register(vlenb), vector start index register(vstart), vector fixed-point rounding mode register(vxrm), vector fixed-point saturation flag(vxsat), vector control and status register(vcsr). Ключевым регистром является регистр vtype. Структура этого регистра изображена на рис. 5.

Конфигурирование системного регистра vtype осуществляется при помощи специальных инструкций vsetvli/vsetivli/vsetvl. Сигнатура инструкций показана на рис. 6.

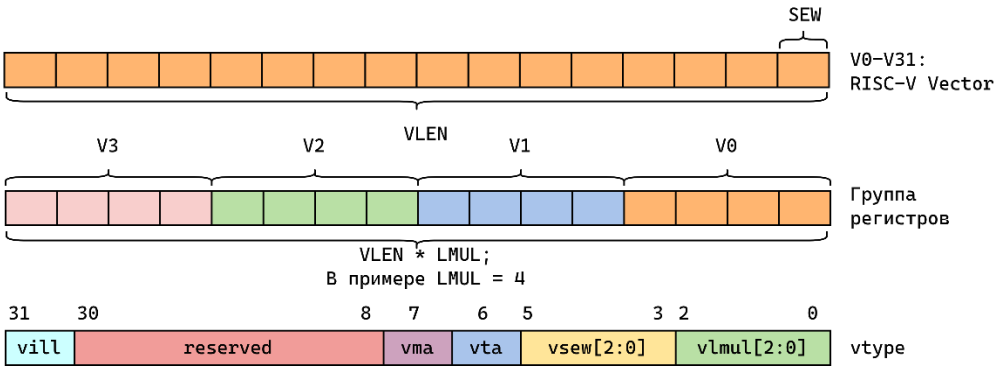


Рис. 5. Векторные регистры расширения RISC-V V; структура системного регистра vtype.
Fig. 5. Vector registers of the RISC-V V extension; structure of the vtype system register;
vill – illegal value if set; vma – vector mask agnostic;.vta – vector tail agnostic;
vsew – selected element width (SEW); vlmul – vector register group multiplier (LMUL).

```
vsetvli rd, rs1, vtypei # rd = new vl, rs1 = AVL, vtypei = new vtype setting
vsetivli rd, uimm, vtypei # rd = new vl, uimm = AVL, vtypei = new vtype setting
vsetvl rd, rs1, rs2 # rd = new vl, rs1 = AVL, rs2 = new vtype value
```

Рис. 6. Сигнатура инструкций конфигурирования системного регистра vtype.
Fig. 6. Signature of instructions configuring the vtype system register.

Инструкции устанавливают значения параметров длины вектора VL, размера элемента вектора SEW и размера группы векторных регистров LMUL. Длина вектора не задается в явном виде. Вместо этого в программе запрашивается необходимая для программы длина вектора (Application Vector Length – AVL). Инструкции vset{i}vl{i} на основе фактической, реализованной аппаратно длины вектора VLEN, а также значений параметров SEW и LMUL вычисляют максимально возможную длину вектора, конфигурируют соответствующим образом регистр vtype и сохраняют вычисленную длину в регистре rd. После этого все векторные инструкции оперируют векторами той длины, которая была сохранена в регистр rd. Регистр rd может быть использован в коде программы, например, для смещения указателей после обработки порции данных. Параметры SEW, LMUL являются операндами инструкций vset{i}vl{i} и могут быть переданы в виде непосредственных констант, закодированных в поле vtypei или через регистр rs2. Значение регистра в этом случае должно содержать эти параметры, закодированные определенным образом.

На рис. 7 приведен пример реализации вычисления saxpy на языке C (Y = a * X + Y) и на ассемблере с использованием инструкций RVV.

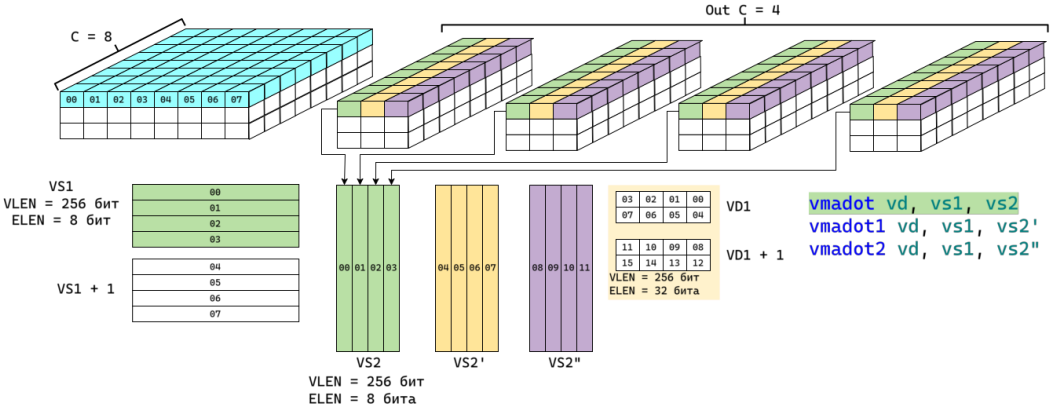
Регистр a0 хранит длину массивов x и y, fa0 – скаляр a, a1 и a2 – адреса начал массивов x и y. В строке 2 выполняется конфигурирование состояния процессора при помощи инструкции vsetvli, в которой указан размер элемента вектора SEW = e32 (элементы по 32 бита), регистры

сгруппированы в векторы по 8 регистров (параметр m8). Доступная длина вектора будет сохранена в регистре a4, а все последующие векторные инструкции будут оперировать векторами с данной длиной. Таким образом, если аппаратная длина VLEN равна 256 битам, то максимальная длина вектора, которая будет сохранена в регистр a4 и записана в VL регистра vtype, будет равна 64 элементам. В строках 3 и 4 происходит загрузка элементов массива x и y в векторы v0 и v8 длиной VL. После этого в строке 5 выполняется инструкция умножения-сложения (a * x + y), результат которой сохраняется в векторном регистре v8. Результат записывается в память в массив y в строке 6. Далее, в строке 7, выполняется уменьшение числа оставшихся для вычисления элементов n на длину вектора VL (которая сохранена в регистре a4). В строке 8 вычисляется длина вектора в байтах. В строках 9-10 смещаются указатели массивов x и y на длину вектора в байтах. В строке 11 выполняется проверка условия выхода из цикла, и управление передается в начало выполнения следующей итерации или на инструкцию выхода из цикла. В случае если остаточная длина массивов n меньше, чем максимальная длина, например 64 элемента, инструкция vsetvli установит параметр VL и значение регистра a4 равным n.

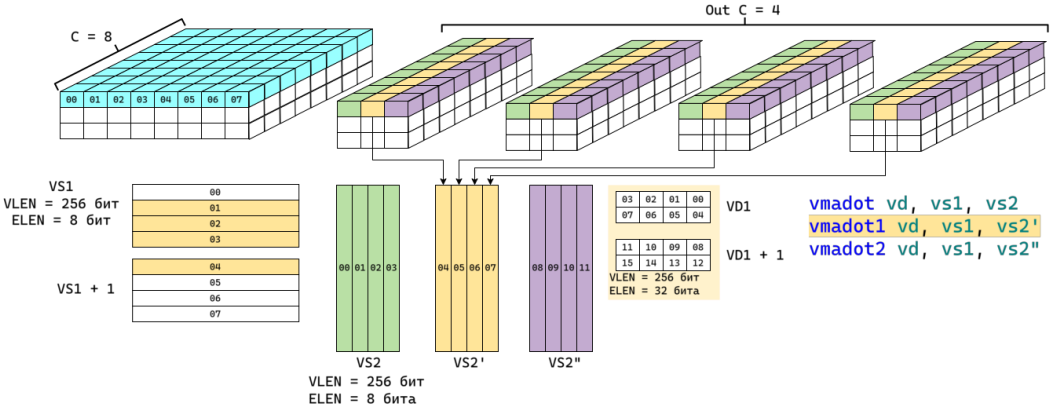
```
# void saxpy(size_t n, const float a,
#            const float *x, float *y) {
#   size_t i;
#   for (i=0; i<n; i++)
#     y[i] = a * x[i] + y[i];
# }
# register arguments:
#     a0      n
#     fa0     a
#     a1      x
#     a2      y
1  saxpy:
2     vsetvli a4, a0, e32, m8, ta, ma
3     vle32.v v0, (a1)
4     vle32.v v8, (a2)
5     vfmacc.vf v8, fa0, v0
6     vse32.v v8, (a2)
7     sub a0, a0, a4
8     slli a4, a4, 2
9     add a1, a1, a4
10    add a2, a2, a4
11    bnez a0, saxpy
12    ret
```

Рис. 7. Пример реализации saxpy на языке C и на ассемблере с использованием RVV инструкций.
Fig. 7. Implementation of saxpy in C and assembly using RVV instructions.

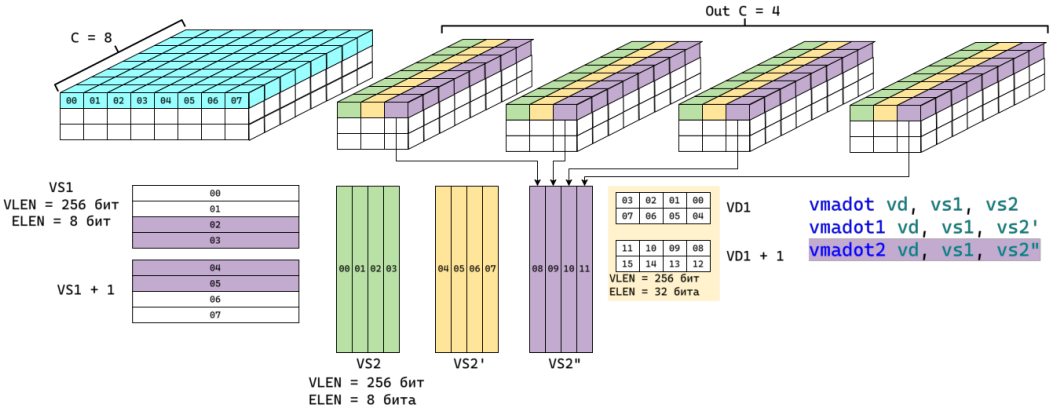
Пример выше демонстрирует основную идею сценария использования RVV, понимание которой необходимо для ознакомления с расширением IME. Расширение RVV представляет одну ключевую инструкцию умножения матриц vmadot (A[M:K] * B[K:N] = C[M:N]), которая может выполняться в двух режимах: в режиме обычного умножения матриц и в режиме «скользящего окна». В режиме «скользящего окна» для умножения из матричного регистра выбирается матрица A со смещением в 1, 2, 3 или n строк. Размеры M, N и K определяются длиной векторных регистров VLEN и выбранным размером элементов вектора SEW [6].



а) выполнение инструкции vmadot vd, vs1, vs2.
a) execution of the vmadot vd, vs1, vs2.



б) выполнение инструкции режима «скользящего окна» vmadot1 vd, vs1, vs2'.
b) execution of the sliding-window vmadot1 vd, vs1, vs2'.



в) выполнение инструкции режима «скользящего окна» vmadot1 vd, vs1, vs2''.
c) execution of the sliding-window vmadot1 vd, vs1, vs2''.

Рис. 8. Пошаговый процесс вычисления фрагмента двумерной свертки с использованием инструкций RISC-V IME; шаг 1 – а; шаг 2 – б; шаг 3 – в.
Fig. 8. Step-by-step process of computing a fragment of a convolution using RISC-V IME instructions; step 1 - a; step 2 - b; step 3 - c.

Режим «скользящего окна» удобен для вычисления операции двумерной свертки методом image2col при помощи операции умножения матриц. Например, если размер входной карты признаков равен 1x3x8x8 (в формате NHWC), а размер ядра свертки – 4x3x3x8 (в формате NHWC), то размер выходной карты признаков будет равен 1x6x6x4 (NHWC). В этом случае применение трех координат каждого из 4 ядер свертки (иначе говоря, одной строки каждого из 4 ядер свертки) может быть вычислено при помощи последовательности из трех инструкций vmadot в режиме «скользящего окна», как показано на рис. 8. Результат выполнения каждой инструкции умножения матриц vmadot будет аккумулироваться в векторном регистре VD.

Расширение RISC-V IME, повторно используя ресурсы регистров векторного расширения, может обеспечить улучшение производительности вычислений искусственных нейронных сетей при очень небольшой стоимости аппаратного обеспечения.

2.3 Матричное расширение RISC-V Attached Matrix Extension

Матричное расширение RISC-V Matrix Multiplication – RVM (Attached Matrix Extension) является независимым матричным расширением процессорной архитектуры RISC-V, которое разрабатывается компанией T-Head (подразделение компании Alibaba). По состоянию на 2024 год расширение RVM находится в разработке. В отличие от интегрированного расширения IME расширение RVM не использует векторные регистры, по этой причине его называют «независимым». Расширение содержит 8 матричных регистров M0-M7, размер которых определяется конкретной реализацией архитектуры. Длина строки каждого матричного регистра составляет RLEN бит, а количество строк равно RLEN / 32. Таким образом, количество столбцов в матричном регистре определяется шириной типа элементов матрицы.

RVM расширение содержит множество инструкций, включающее в себя: инструкции умножения матриц, инструкции загрузки матричных регистров из памяти и сохранения их значений в память, инструкции поэлементных операций над матричными регистрами (поэлементное сложение, умножение, вычитание и др.), инструкции перемещения данных между матричными регистрами, инструкции для обнуления матричных регистров, служебные инструкции конфигурирования размера матричного блока.

2.4 Архитектура DaVinci

Процессорная архитектура DaVinci [7] разработана компанией HiSilicon и ориентирована на решение задач, связанных с выполнением и обучением искусственных нейронных сетей. В 2018 году был анонсирован процессор Ascend AI, реализующий данную архитектуру. Архитектура DaVinci является закрытой, однако в открытых источниках опубликовано некоторое количество работ, позволяющих получить представление о принципах ее организации, а также некоторых особенностях ее функционирования.

На рис. 9 представлена функциональная структура процессорного ядра AI core с архитектурой DaVinci.

Ядро с архитектурой DaVinci содержит три основных вычислительных процессорных элемента: скалярный (Scalar Unit), векторный (Vector Unit) и кубический (Cube Unit) процессорные элементы. Скалярный процессорный элемент выполняет арифметические и логические операции над скалярными переменными, а также выполняет команды, определяющие поток управления (команды условной и безусловной передачи управления). Векторный процессорный элемент выполняет команды, реализующие арифметические и логические операции над векторами данных. Кубический процессорный элемент выполняет команды, реализующие операцию умножения матриц фиксированного размера, и по своей сути является систолическим массивом АЛУ.

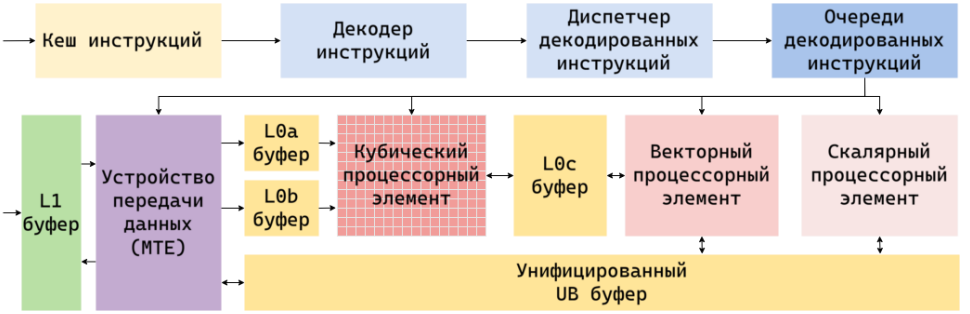


Рис. 9. Функциональная структура архитектуры DaVinci.
Fig. 9. The structure of the DaVinci architecture.

Кроме вычислительных процессорных элементов архитектура DaVinci обладает устройствами передачи данных (Memory Transfer Element – MTE), которые выполняют копирование данных между различными типами памяти. Одновременно с копированием может быть выполнена обработка данных, например преобразование типов из fp32 в fp16, транспонирование копируемого блока данных, вычисление функции активации (например, ReLU) и другие, более сложные, преобразования формата хранения данных.

Архитектура DaVinci реализует неоднородную иерархическую память трех типов: буфер L1, буфер L0 и унифицированный буфер (Unified Buffer – UB). Буфер L0 предназначен для хранения операндов кубического процессорного элемента, то есть умножаемых матриц A и B. Матрица A должна быть размещена в буфере L0a, а матрица B – в L0b. Матрицы должны быть сохранены в блочном формате. Размер блока зависит от типа элементов матриц, например для типа fp16 размер блока составляет 16x16 элементов. Результат умножения матриц сохраняется в буфере L0c также в блочном формате. Унифицированный буфер UB хранит операнды векторных инструкций. Буфер L1 служит для временного размещения часто используемых входных данных, а также для сохранения промежуточных результатов вычислений. Обычный сценарий организации вычислений на ядре с процессорной архитектурой DaVinci включает в себя следующие манипуляции с данными:

1) копирование входных параметров в буфер L1;
2) копирование фрагмента входных данных из буфера L1 в буферы L0a и L0b;
3) вычисление произведения матриц на кубическом процессорном элементе с аккумулированием результата в буфере L0c;
4) копирование результата умножения матриц из буфера L0c в унифицированный буфер UB;
5) обработка данных, скопированных в унифицированный буфер UB, на векторном процессорном элементе;
6) копирование данных из унифицированного буфера UB в буфер L1 для сохранения промежуточных результатов вычислений.

Каждый вычислительный процессорный элемент и устройство передачи данных обладает собственной очередью инструкций. После декодирования инструкций специальное устройство, называемое диспетчером декодированных инструкций, помещает очередную инструкцию в соответствующую очередь. Выполнение инструкций из каждой очереди организовано по принципу FIFO (первым пришел – первым обслужен). Таким образом процессорные устройства и устройства передачи данных могут функционировать одновременно и независимо друг от друга, обеспечивая параллелизм уровня инструкций. Синхронизация функционирования процессорных элементов и устройств передачи данных

осуществляется посредством размещения в соответствующих очередях специальных инструкций. Эти инструкции моделируют обмен сообщениями, содержащими события синхронизации. Иначе говоря, в очередь одного устройства вставляется инструкция отправки события синхронизации, а в очередь другого – инструкция приема этого события. При выполнении инструкции приема устройство блокируется до тех пор, пока не будет получено соответствующее событие синхронизации, т.е. пока другое устройство не выполнит инструкцию отправки.

Кубический процессорный элемент позволяет эффективно выполнять операции умножения матриц. Эффективное выполнение операции двумерной свертки на вычислительном ядре с архитектурой DaVinci возможно путем выражения операции двумерной свертки через операции умножения матриц методом im2col. Так как входные данные для кубического процессорного элемента должны храниться в памяти в блочном формате, то для вычисления двумерной свертки посредством умножения матриц необходимо изменить формат хранения операндов свертки соответствующим образом. Архитектура DaVinci предоставляет специальные инструкции, выполняющие такое преобразование формата многомерных массивов, выполняемое на одном из устройств передачи данных.

Резюмируя выше сказанное, архитектура DaVinci содержит инструкции для выполнения умножения матриц, хранящихся в памяти в блочном формате, а также инструкции преобразования формата хранения многомерных массивов im2col. Ввиду того, что архитектура является закрытой, у авторов данной работы нет возможности продемонстрировать пример реализации умножения матриц с использованием кубических инструкций. Однако, для проектирования абстрактного высокоуровневого представления значимым является лишь наличие в архитектуре матричных инструкций.

2.5 Модель матричного процессора

В результате проведенного анализа матричных расширений и поддерживающих матричные инструкции процессорных архитектур были выявлены следующие основные аспекты:

- наборы матричных инструкций оперируют набором матричных регистров или специальной памятью для хранения их операндов;
- матричные регистры хранят матричные блоки;
- матричные регистры могут использовать векторные регистры или формировать независимый регистровый файл;
- для эффективного использования матричных инструкций требуется блочное хранение матриц;
- матричные процессорные устройства могут не иметь возможности напрямую обращаться к оперативной памяти.

Таким образом, проведенный анализ позволяет предложить следующую модель абстрактной матричной процессорной архитектуры. На рис. 10. представлена общая схема модели.

Модель включает в себя 4 адресных пространства: 1) адресное пространство управляющего устройства; 2) адресное пространство внешнего вычислительного устройства; 3) матричное адресное пространство; 4) векторное адресное пространство. *Адресное пространство управляющего устройства* является абстракцией над оперативной памятью. Предполагается, что данное адресное пространство содержит данные для скалярных и служебных вычислений. *Матричные* и *векторные адресные пространства* служат для хранения операндов матричных и векторных процессорных устройств. *Адресное пространство внешнего вычислительного устройства* является абстракцией над памятью сопроцессора, в случае если матричный процессор реализован в виде отдельного устройства, например память GPU или AI ускорителя (или High Bandwidth Memory – HBM память). Когда матричный процессор реализован в виде специального АЛУ процессорного ядра, это

адресное пространство может отсутствовать или являться абстракцией процессорного кэша высокого уровня.

Предлагаемая модель также содержит следующие абстрактные вычислительные устройства: 1) процессор управляющего устройства; 2) матричное устройство; 3) векторное устройство.

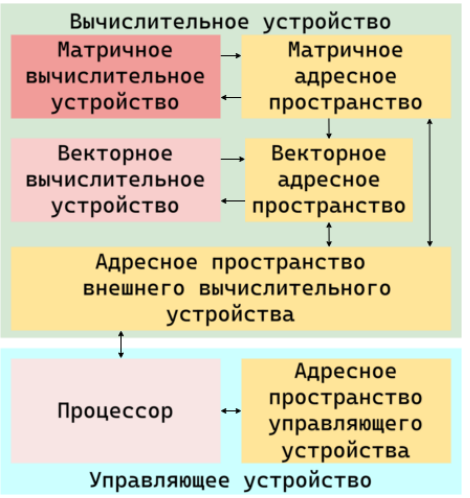


Рис. 10. Модель матричного процессора.
Fig. 10. Model of a matrix processor.

3. Компилятор моделей машинного обучения

Настоящий раздел содержит описание предлагаемого подхода к реализации компиляторов искусственных нейронных сетей для процессорных архитектур, которые поддерживают матричные инструкции. Предлагаемый подход ориентируется на введенную в предыдущем разделе модель абстрактной матричной архитектуры. Для построения компилятора предлагается семейство многоуровневых промежуточных представлений. Реализация представлений может быть выполнена в инфраструктуре MLIR. Далее раскрываются детали предлагаемого семейства промежуточных представлений, а также использование инфраструктуры MLIR для их реализации.

3.1 Обзор инфраструктуры MLIR

Проект многоуровневого промежуточного представления (Multiple Level Intermediate Representation – MLIR) является частью проекта LLVM. Основная цель проекта состоит в предоставлении необходимой единой инфраструктуры для проектирования и разработки промежуточных представлений. Такая инфраструктура, обычно, включает в себя менеджер проходов, движок сопоставления с образцом (поиск соответствия шаблонам) и интерфейсы для трансляции инструкции одних промежуточных представлений в инструкции других и т. д. Тем самым для существующих и вновь создаваемых компиляторов исчезнет необходимость в разработке такой инфраструктуры, а разработчики могут сосредоточить свои усилия на разработке оптимизирующих трансформаций и на проектировании новых промежуточных представлений. MLIR представляет такую инфраструктуру, а также содержит множество встроенных промежуточных представлений.

Инфраструктура MLIR не содержит конкретных инструкций, вместо этого она предоставляет необходимые абстракции для разработки произвольных промежуточных представлений, которые обладают иерархической структурой, например, как у LLVM IR: модуль содержит функции, функции содержат базовые блоки, базовые блоки содержат инструкции.

Представление MLIR состоит из трех ключевых сущностей: 1) операция; 2) регион; 3) блок. Операция – единица, задающая семантику выполнения. Операция может содержать регион. Регион – это контейнер для операций. Регион содержит блоки, которые содержат операции. Блок – это линейная последовательность операций (аналогична базовому блоку). Последняя операция каждого блока содержит операцию передачи управления другому блоку или родительскому региону. Таким образом, представление MLIR обладает иерархической рекурсивной структурой: операция содержит регион, регион содержит блоки, каждый блок содержат операции и т.д.

На рис. 11 представлен пример рекурсивной структуры MLIR-представления.

В этом примере операция builtin.module содержит один регион (в фигурных скобках), в котором неявно содержится один блок. Этот блок включает в себя одну операцию func.func,содержащую один регион, который в свою очередь содержит один явный блок ^BB0. Блок ^BB0 содержит последовательность операций arith.constant и операцию scf.for, которая, в свою очередь, включает в себя регион с одним блоком, содержащем операции memref.load, arith.mulf и т. д. Представление MLIR состоит из трех основных объектов: 1) значения; 2) типы; 3) атрибуты. Значениями являются SSA-имена переменных. В примере на рис. 11 значениями являются %[0-1] и %arg[0-3]. Все значения должны быть типизированными (f32, memref<128xf32>, ...). Атрибуты содержат дополнительную информацию о MLIR-сущностях, например атрибут fastmath = #arith.fastmath<none>.

```
"builtin.module"() ({
  "func.func"() <{
    function_type = (f32, memref<128xf32>, memref<128xf32>) → (),
    sym_name = "saxpy"}> ({
  ^bb0(%arg0: f32, %arg1: memref<128xf32>, %arg2: memref<128xf32>):
    %0 = "arith.constant"() <{value = 0 : index}> : () → index
    %1 = "arith.constant"() <{value = 1 : index}> : () → index
    %2 = "arith.constant"() <{value = 128 : index}> : () → index
    "scf.for"(%0, %2, %1) ({
    ^bb0(%arg3: index):
      %3 = "memref.load"(%arg1, %arg3)
        : (memref<128xf32>, index) → f32
      %4 = "arith.mulf"(%arg0, %3)
        <{fastmath = #arith.fastmath<none>}>
        : (f32, f32) → f32
      %5 = "memref.load"(%arg2, %arg3)
        : (memref<128xf32>, index) → f32
      %6 = "arith.addf"(%4, %5)
        <{fastmath = #arith.fastmath<none>}>
        : (f32, f32) → f32
      "memref.store"(%6, %arg2, %arg3)
        : (f32, memref<128xf32>, index) → ()
      "scf.yield"() : () → ()
    }) : (index, index, index) → ()
    "func.return"() : () → ()
  }) : () → ()
}) : () → ()
```

Рис. 11. Рекурсивная структура MLIR-представления.
Fig. 11. Recursive structure of the MLIR representation.

Инфраструктура MLIR не содержит фиксированного набора инструкций, она лишь предоставляет необходимые инструменты для разработки собственного набора операций, типов и атрибутов. Объединение множества операций, типов и атрибутов формирует диалект. Диалект может быть представлен как библиотека, которая содержит необходимые структуры разработанного промежуточного представления. MLIR содержит множество встроенных диалектов (48 диалектов) для различных сценариев использования: диалект с задающими поток управления операциями; диалект арифметических операций, диалект

аффинных циклов, диалект векторных операций, диалект архитектурно-ориентированных операций и др. Одной из ключевых возможностей инфраструктуры MLIR является определение пользовательских диалектов на специальном предметно-ориентированном языке. Более того, MLIR позволяет описывать в декларативной форме шаблоны для сопоставления с образцом и перезаписи. В следующих разделах содержится описание функциональной структуры компилятора и особенности реализации гетерогенного матричного промежуточного представления.

3.2 Гетерогенное матричное промежуточное представление

Предлагаемая в работе модель матричного процессора содержит процессорные элементы для вычисления матричных и векторных инструкций и несколько типов адресных пространств. Следовательно, гетерогенное матричное промежуточное представление должно содержать необходимые операции для организации вычислений на каждом процессорном элементе, операции копирования данных между различными адресными пространствами, а также служебные операции для диспетчеризации матричных и векторных операций. Служебные операции выполняются вычислительным устройством общего назначения. На нем же выполняются операции копирования данных между адресными пространствами. Так как эффективное использование матричных инструкций требует блочного хранения матриц в памяти, то предлагаемое множество операций содержит соответствующие операции.

В результате был разработан диалект heteroMx, содержащий необходимый набор вычислительных и служебных операций. При описании диалекта используются следующие обозначения. Адресное пространство управляющего устройства (host address space – has) имеет идентификатор, равный 0, и соответствует адресному пространству процесса. Адресное пространство внешнего устройства (device address space – das) имеет идентификатор 1. Матричное адресное пространство (matrix address space – mas) имеет идентификатор 2. Векторное адресное пространство (vector address space – vas) имеет идентификатор 3. Предлагаемый диалект содержит набор операций, представленный в табл. 2.

Реализация операции изменения формата хранения матриц во многом полагается на доступные в целевой архитектуре инструкции. Как правило, удобные для этих целей инструкции содержатся в векторном расширении. Эти инструкции реализуют все различные перестановки элементов в векторных регистрах. В худшем случае эта операция может породить скалярный код, в котором матрица сохраняется в блочном формате поэлементно.

Необходимо отметить, что операции диалекта heteroMx работают как на уровне тензорной семантики, так и на уровне семантики многомерных массивов в памяти. Тензорная семантика позволяет абстрагироваться от адресных пространств конкретного типа и памяти как таковой. Тензоры – это абстрактные N-мерные агрегаты с известным типом элементов и известным рангом. Семантику многомерных массивов в памяти в терминах MLIR задают ссылки на области памяти (memref). Процесс перехода от тензорной семантики к семантике многомерных массивов в памяти называется буферизацией. Инфраструктура MLIR содержит все необходимые примитивы для автоматизации этого процесса, однако часть действий необходимо реализовать на стороне разрабатываемого компилятора, например, выделение буферов под тензоры в адресных пространствах, вычисление размещения буферов в адресных пространствах и др. Эти вопросы не будут раскрыты в данной работе, а являются темой будущих исследований. Описание диалекта, а также его операций выполняется в декларативной форме.

Предлагаемый подход к построению компилятора использует доступные в инфраструктуре MLIR диалекты. К числу таких диалектов относятся диалекты крупноблочных операций нейронных сетей, диалект для организации аффинных гнезд циклов, диалект emitc и др. В следующем разделе представлено описание общей структуры компилятора.

Табл. 2. Операции диалекта heteroMx.
Table 2. Operations of the heteroMx dialect.

heteroMx.mad a:Tensor<*>, b:Tensor<*>, c:Tensor<*>	Операция умножения матриц c = a × b
heteroMx.blocking src:Tensor<*>, dst:Tensor<*>	Операция, преобразующая формат хранения матриц в блочный
heteroMx.unblocking src:Tensor<*>, dst:Tensor<*>	Операция, восстанавливающая исходный формат хранения матриц
heteroMx.element_wise_(add/sub/mul/div/sqrt/max/min/and/or/rsqrt/relu/...) src1:Tensor<*>, src2:Tensor<*>, dst:Tensor<*>	Поэлементные векторные операции
heteroMx.reduce_(sum/prod/min/max) src:Tensor<*>, dst:Tensor<*>	Операция вычисления свертки вектора
heteroMx.zero dst:Tensor<*>	Операция очистки операнда. Операнд может быть размещен во всех доступных адресных пространствах
heteroMx.copy_has_to_das src:Tensor<*>, dst:Tensor<*> heteroMx.copy_das_to_has * heteroMx.copy_das_to_mas * heteroMx.copy_das_to_vas * heteroMx.copy_mas_to_das * heteroMx.copy_vas_to_das * heteroMx.copy_vas_to_mas * heteroMx.copy_mas_to_vas *	Операции копирования соответствующих буферов памяти
heteroMx.kernel_run @func_name	Операция запуска функции, которая является вычислительным ядром на матричном процессоре

3.3 Функциональная структура компилятора

Компилятор моделей машинного обучения (AI-компилятор) в качестве входной программы принимает описание вычислительного графа модели на DSL-языке в одном из возможных форматов, например ONNX, PyTorch, TensorFlow и др. На данном этапе могут быть применены преобразования модели, специфичные для конкретной библиотеки машинного обучения. Вычислительный граф модели должен быть преобразован в программу на соответствующем диалекте. Для большинства популярных библиотек машинного обучения существуют соответствующие диалекты. Так проект onnx-mlir реализует onnx диалект, torch-mlir – torch диалект и т.д. Также существуют трансляторы из исходного кода DSL-языка в операции соответствующего диалекта.

Следующим шагом необходимо транслировать модель в высокоуровневый диалект, независимый от конкретной библиотеки машинного обучения. Диалект на данном уровне содержит крупноблочные операции, характерные для искусственных нейронных сетей:

операции двумерной свертки, умножения матриц и т.д. Существуют сторонние проекты, которые реализуют диалекты с крупноблочными операциями, например stablehlo, mhlo, chlo диалекты. Однако инфраструктура MLIR содержит стандартизированный набор крупноблочных тензорных операций в TOSA [8] диалекте (Tensor Operator Set Architecture – TOSA). Предлагаемый AI-компилятор использует данный диалект. На рис. 12 изображена функциональная структура AI-компилятора.

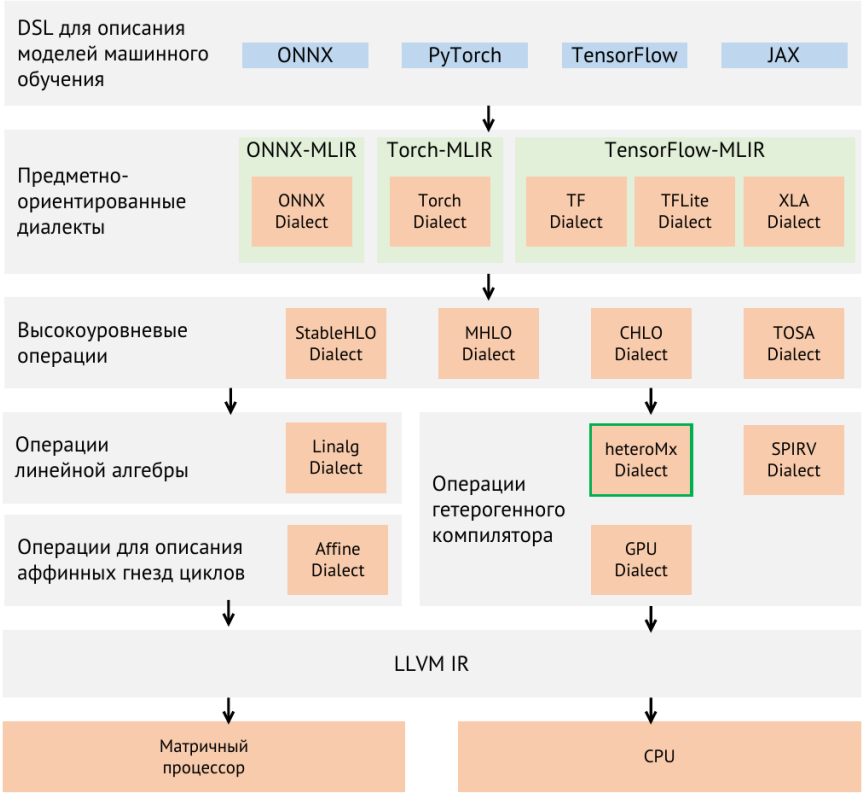


Рис. 12. Функциональная структура AI-компилятора.
Fig. 12. The structure of the AI compiler.

На уровне TOSA диалекта могут выполняться простые оптимизации для сокращения избыточных операций, например избыточные транспонирования вида transpose(transpose(A) = A. MLIR позволяет описать подобные преобразования в декларативной форме на специальном языке. На рис. 13 продемонстрирован пример описания такого преобразования.

```
/// transpose(transpose(in)) → in
def TransposeElimination : Pat<
  (Tosa_TransposeOp (Tosa_TransposeOp $in, $perms_in), $perms_out),
  (replaceWithValue $in),
  [(Constraint<CPred<"$0 == $1">> $perms_in, $perms_out)]>;
```

Рис. 13. Описание шаблона для устранения избыточных операций транспонирования.
Fig. 13. Description of the pattern for eliminating redundant transpose operations.

Следующим шагом TOSA-программа транслируется в linalg диалект, который содержит операции линейной алгебры. Часть операций, такие, как умножение матриц и двумерные свертки, транслируются в операции разработанного диалекта heteroMx. После этого

программа транслируется в диалект affine, и в ней появляются аффинные циклы. На уровне операций диалекта affine, а также в процессе трансляции в операции разработанного диалекта heteroMx, выполняется разбиение циклов на блоки циклов (tiling циклов), а также матриц (в терминах работы Goto и Geijn [9] – упаковка блоков матриц). Наличие явных аффинных циклов позволяет применять полиэдральные преобразования циклов.

После этого программа, представленная в диалектах affine и heteroMx, поэтапно, прогрессивно (то есть с малым шагом интерпретации) транслируется в LLVM IR и далее в код целевой архитектуры матричного процессора. На данном этапе могут быть использованы архитектурно-ориентированные диалекты, например, arm_sme для использования инструкций матричного расширения архитектуры ARM или SPIRV для генерации вычислительных ядер гетерогенной архитектуры или другие диалекты. Однако, этот этап трансляции находится за рамками данной работы и, несомненно, является важнейшим, с большим количеством открытых научных вопросов.

Результатом компиляции, при таком подходе, может быть LLVM код (совместно с SPIRV кодом) или машинный код, в случае если трансляция шла по вышеобозначенному пути. На этапе, когда программа представлена в диалектах affine и heteroMx, возможен альтернативный путь. Программа может быть транслирована в emitc диалект, и тогда результирующим артефактом будет программа на языке C++, которая может быть скомпилирована в код целевой архитектуры любым доступным компилятором.

4. Заключение

Развитие матричных расширений процессорных архитектур, а также внедрение этих расширений в специализированные AI-процессоры, позволяет существенно повысить эффективность выполнения искусственных нейронных сетей. В работе выполнен обзор базовых функциональных возможностей некоторых популярных матричных расширений процессорных архитектур, в частности расширений ARM SME, RISC-V IME, RISC-V AME, а также процессорной архитектуры DaVinci. В результате анализа нами была предложена модель абстрактного матричного процессора, которая отражает особенности современных процессорных архитектур, поддерживающих матричные расширения.

Для введенной модели матричного процессора нами разработано гетерогенное матричное промежуточное представление. Предложенное промежуточное представление может быть использовано для построения компиляторов нейронных сетей. Работа затрагивает только промежуточные представления высокого уровня, представления среднего и низкого уровня остаются в качестве направлений для дальнейших исследований. Предложенное промежуточное представление нами было реализовано в инфраструктуре MLIR в виде диалекта heteroMx. В работе описан подход к построению AI-компилятора с использованием разработанного диалекта heteroMx. Гетерогенное матричное промежуточное представление может быть адаптировано или конкретизировано для других матричных процессорных архитектур, которые не были рассмотрены в настоящей работе, но имеют сходства с предложенной моделью.

Предложенное промежуточное представление и разработанный диалект могут быть использованы при реализации гетерогенного компилятора, например, в качестве высокоуровневого преставления в компиляторе, реализующем стандарт SYCL для матричной целевой архитектуры.

Список литературы / References

[1]. Sousa R. et al. Tensor slicing and optimization for multicore NPUs //Journal of Parallel and Distributed Computing. – 2023. – Т. 175. – С. 66-79.

[2]. Jouppi N. P. et al. In-datacenter performance analysis of a tensor processing unit //Proceedings of the 44th annual international symposium on computer architecture. – 2017. – С. 1-12.

[3]. Lattner C. et al. MLIR: Scaling compiler infrastructure for domain specific computation //2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). – IEEE, 2021. – С. 2-14.
[4]. Remke S., Breuer A. Hello SME! Generating Fast Matrix Multiplication Kernels Using the Scalable Matrix Extension //arXiv preprint arXiv:2409.18779. – 2024.
[5]. Stephens N. et al. The ARM scalable vector extension //IEEE micro. – 2017. – Т. 37. – №. 2. – С. 26-39.
[6]. The RISC-V IME Set Specification. https://github.com/space-mit/riscv-ime-extension-spec/releases/download/v0429/spacemit-ime-asciidoc.pdf.
[7]. H. Liao, J. Tu, J. Xia and X. Zhou, "DaVinci: A Scalable Architecture for Neural Network Computing," 2019 IEEE Hot Chips 31 Symposium (HCS), Cupertino, CA, USA, 2019, pp. 1-44, doi: 10.1109/HOTCHIPS.2019.8875654.
[8]. TOSA specification. https://www.mlplatform.org/tosa/tosa_spec.html, Accessed July 2023.
[9]. Goto K., Geijn R. A. Anatomy of high-performance matrix multiplication //ACM Transactions on Mathematical Software (TOMS). – 2008. – Т. 34. – №. 3. – С. 1-25.

Информация об авторах / Information about authors

Иван Иванович КУЛАГИН – кандидат технических наук, научный сотрудник ИСП РАН. Область научных интересов: построение компиляторов, оптимизирующие компиляторы, полиэдральная компиляция, генерация кода, модели параллельного программирования, AI-ускорители.

Ivan Ivanovich KULAGIN – Cand. Sci. (Tech.), Researcher in ISP RAS. Research interests: compiler construction, compiler optimizations, polyhedral compilation, code generation, parallel programming models, AI-accelerators.

Рубен Артурович БУЧАЦКИЙ – кандидат технических наук, научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Cand. Sci. (Tech.), researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Михаил Вячеславович ПАНТИЛИМОНОВ – научный сотрудник отдела компиляторных технологий. Научные интересы: статический анализ, компиляторные технологии, СУБД.

Mikhail Vyacheslavovich PANTILIMONOV – researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, DBMS.

Андрей Викторович ВЯЗОВЦЕВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Andrey Viktorovich VYAZOVTSEV – a student at MIPT, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Михаил Максимович РОМАНОВ – студент ВМК МГУ, лаборант отдела компиляторных технологий ИСП РАН. Сфера научных интересов: компиляторные технологии, ускорение искусственных нейронных сетей.

Mikhail Maksimovich ROMANOV – a student of CMC department of MSU, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: compiler technologies, artificial neural networks acceleration.

Дмитрий Михайлович МЕЛЬНИК – старший научный сотрудник отдела компиляторных технологий Института системного программирования с 2004 года. Сфера научных интересов: компиляторные оптимизации, динамическая (JIT) компиляция.

Dmitry Mikhailovich MELNIK – Senior Researcher in Compiler Technology department. Research interests: compiler optimizations, dynamic (JIT) compilation.