АЛЬЯНС RISC-V

ПЕРВЫЙ МИТАП АЛЬЯНСА RISC-V

При поддержке YADRO

topic:

Матричные расширения RISC-V:
где, когда, куда, откуда,
почему, зачем и как?;

```
.speaker {
  name:     Валерия Пузикова;
  position: YADRO;
}
```

Валерия Пузикова

К.ф.-м.н., эксперт по разработке ПО, YADRO

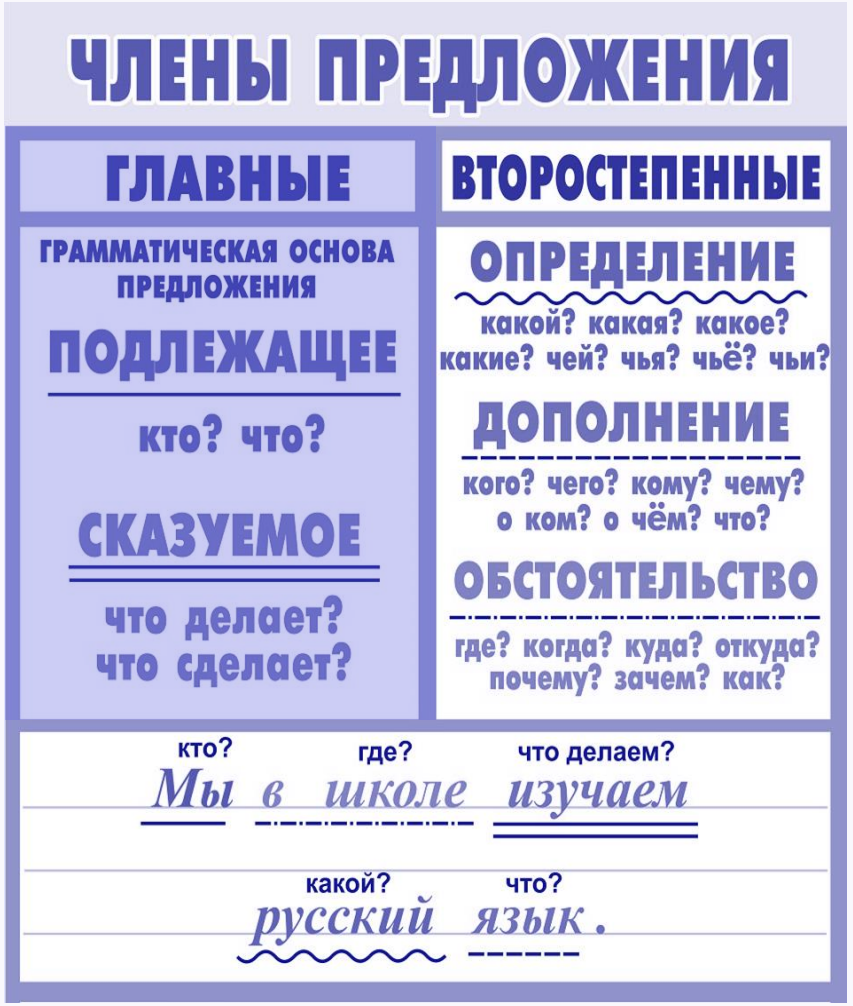
- С 2010 года разрабатываю и реализую на C/C++ с CUDA/MPI/OpenMP численные методы для решения задач линейной алгебры, вычислительной аэрогидродинамики, AR/VR.
- Работала в Huawei, Fortum, ИСП РАН им. В.П. Иванникова, МГТУ им. Н.Э. Баумана и др.

Что это? Что оно делает?

Матричные расширения ISA CPU позволяют ускорять операции над матрицами (в первую очередь умножение) без использования отдельных ускорителей.

Что? Что делает?

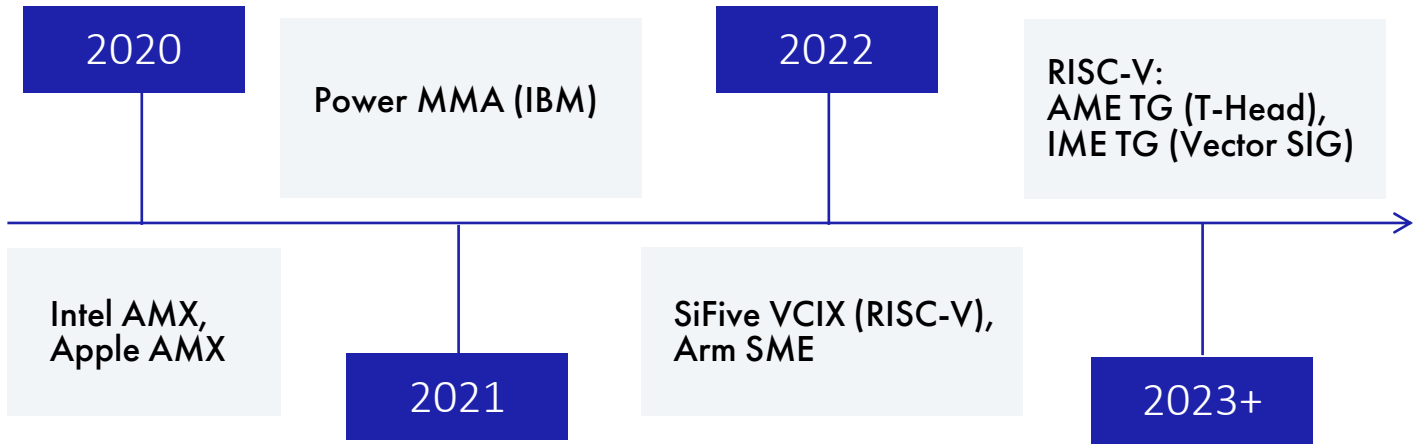
Матричные расширения ISA CPU позволяют ускорять операции над матрицами (в первую очередь умножение) без использования отдельных ускорителей.

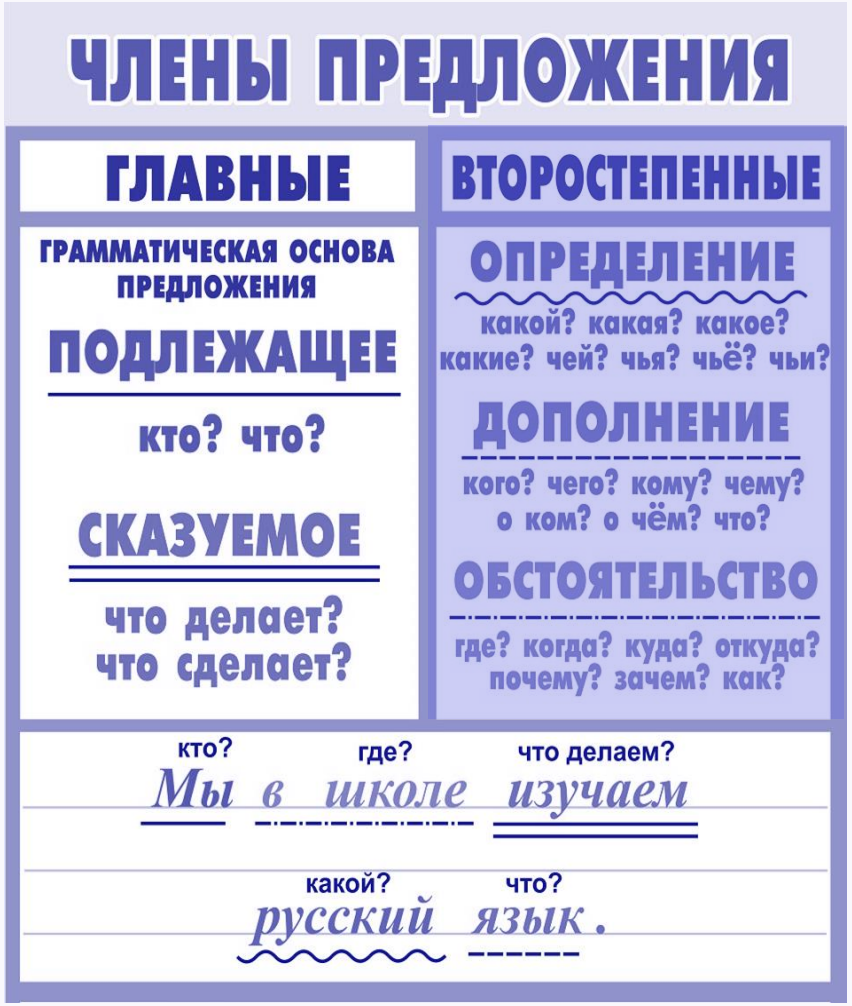


* Источник: https://infodoski.ru/images/detailed/34/36980.jpg

Где? Когда?

С 2020 года: x86, Arm, Power, RISC-V.





* Источник: https://infodoski.ru/images/detailed/34/36980.jpg

Определения и обстоятельства

Разработка T-Head RVM ⟶ RISC-V AME TG

Предложения RISC-V Vector SIG ⟶ RISC-V IME TG

Свежие вести с полей: Sparse Lives Matter

Заключение

Зачем?

Основная нагрузка – матричные операции:

- умножение плотных матриц (GEMM);
- умножение разреженной матрицы на вектор;
- прочие операции в разных пропорциях в зависимости от конкретных приложений.

AI/ML – ИИ и машинное обучение.
CV – компьютерное зрение.
AR/VR – дополненная и виртуальная реальность.
ADAS – система помощи водителю.
HPC – математическое моделирование.

Зачем?

Основная нагрузка – матричные операции:

- умножение плотных матриц (GEMM);
- умножение разреженной матрицы на вектор;
- прочие операции в разных пропорциях в зависимости от конкретных приложений.

Развитие технологий в этих областях увеличивает спрос на вычислительные мощности на порядки.

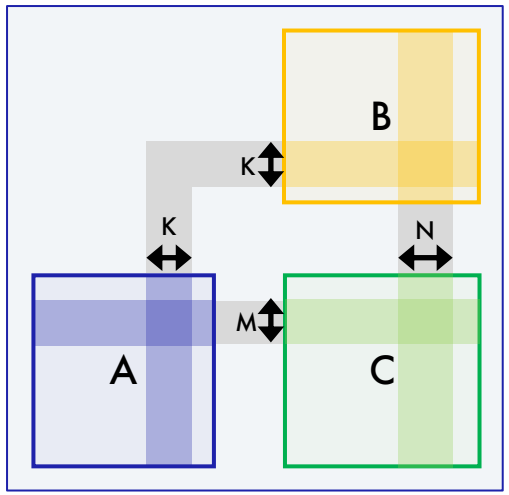
Нужно ускоряться...

А как же GPU/TPU/...?

Ускорители – это прекрасно, но жизнь такова, что и на CPU тоже надо ускоряться...

- Доля матричных нагрузок велика во всех сегментах оборудования;
- Ускоритель может быть уже занят другими задачами;
- Даже в сегменте суперкомпьютеров есть гомогенные машины:
  - Fujitu Fugaku (Arm A64FX)
    - №2 в TOP500,
    - №1 в HPCG, HPL-AI, Graph500.

Как ускоряться без ускорителей и расширений?



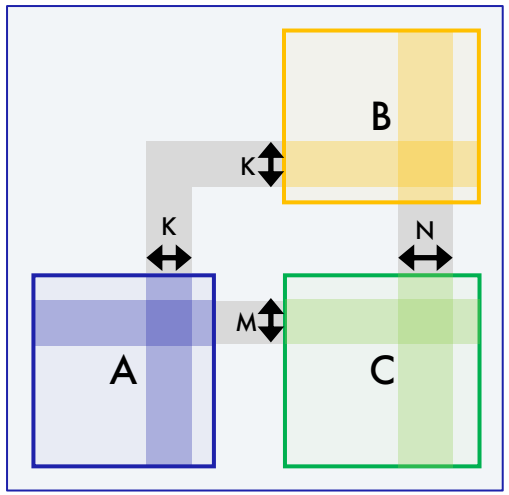
Например, рассмотрим **умножение матриц**

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}.$$

Это $M \cdot N$ скалярных произведений строк первого операнда на столбцы второго:

$$\begin{bmatrix} C_0^0 & \cdots & C_0^{N-1} \\ \vdots & \ddots & \vdots \\ C_{M-1}^0 & \cdots & C_{M-1}^{N-1} \end{bmatrix} = \begin{bmatrix} \sum_{k=0}^{K-1} A_0^k B_k^0 & \cdots & \sum_{k=0}^{K-1} A_0^k B_k^{N-1} \\ \vdots & \ddots & \vdots \\ \sum_{k=0}^{K-1} A_{M-1}^k B_k^0 & \cdots & \sum_{k=0}^{K-1} A_{M-1}^k B_k^{N-1} \end{bmatrix}.$$

Как ускоряться без ускорителей и расширений?



Например, рассмотрим **умножение матриц**

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}.$$

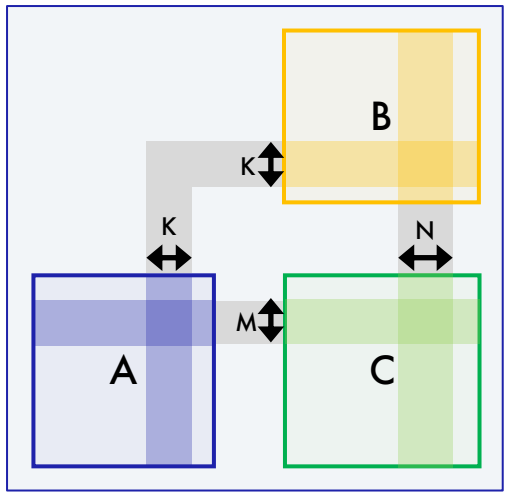
Алгоритм аккумуляции произведения очень прост:

$$C_{M\times N} = [0]_{M\times N}, \qquad \text{for i} = [0, M)$$
$$\text{for j} = [0, N)$$
$$\text{for k} = [0, \text{K}) \;\; C_i^j \mathrel{+}= A_i^k B_k^j.$$

Это $M \cdot N \cdot K$ умножений-сложений.

Число операций не сократить...

Как ускоряться без ускорителей и расширений?



Например, рассмотрим **умножение матриц**

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}.$$

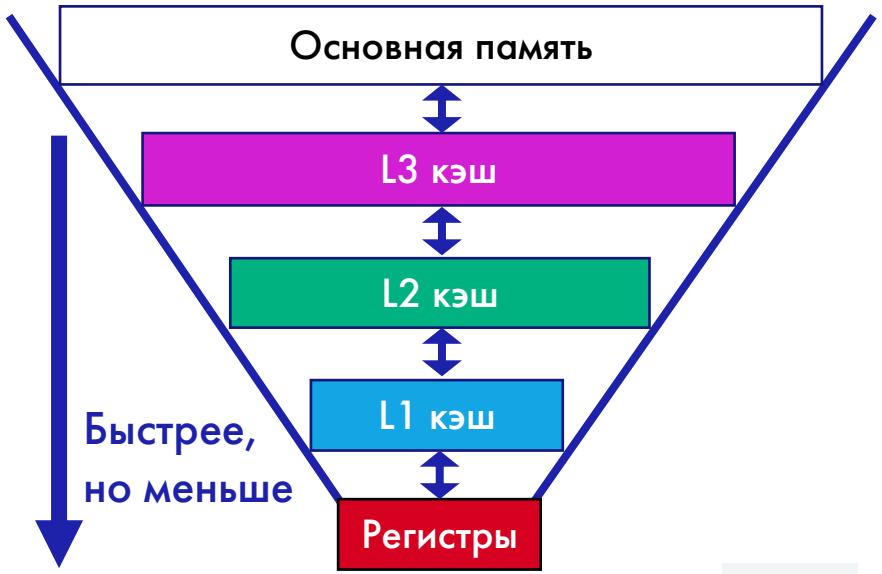
Алгоритм аккумуляции произведения очень прост:

$$C_{M\times N} = [0]_{M\times N}, \qquad \text{for i} = [0, M)$$
$$\text{for j} = [0, N)$$
$$\text{for k} = [0, \text{K}) \;\; C_i^j \mathrel{+}= A_i^k B_k^j.$$

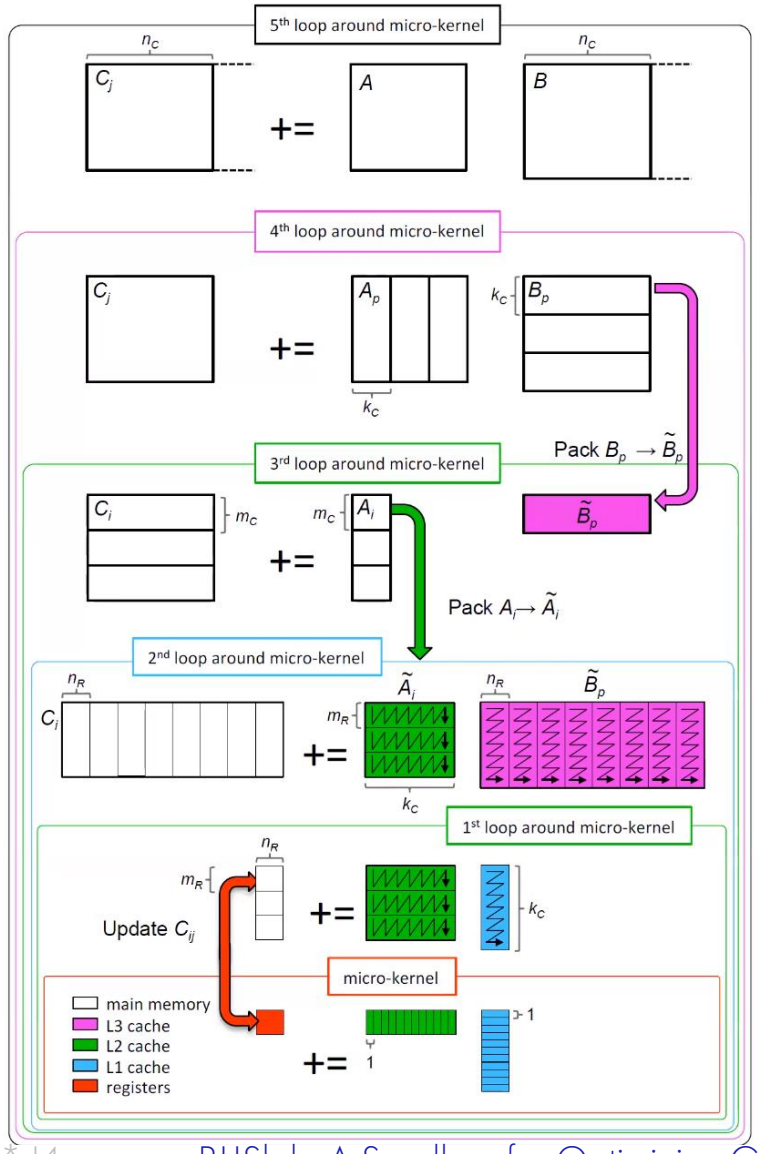
Это $M \cdot N \cdot K$ умножений-сложений.

Число операций не сократить...

Но скорость их выполнения может быть разной!



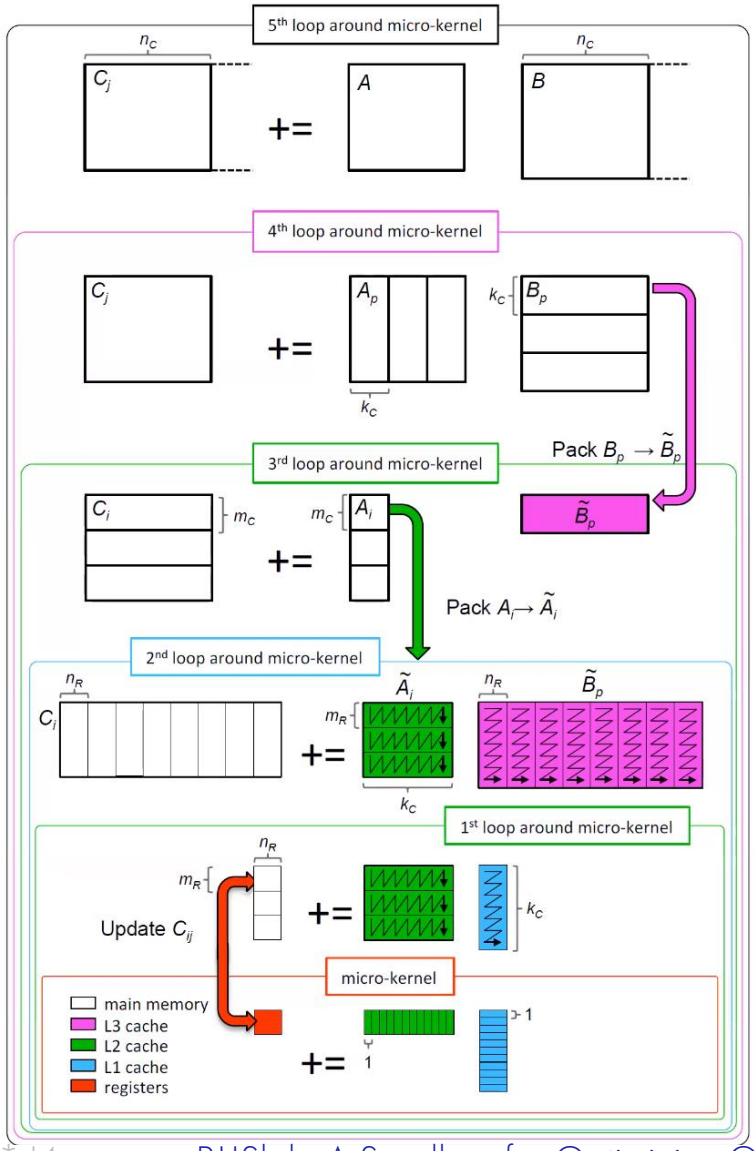
Goto,Geijn: Anatomy of High-Performance Matrix Multiplication



- Этот подход используется в библиотеках OpenBLAS и BLIS.

* Источник: BLISlab: A Sandbox for Optimizing GEMM

Goto,Geijn: Anatomy of High-Performance Matrix Multiplication

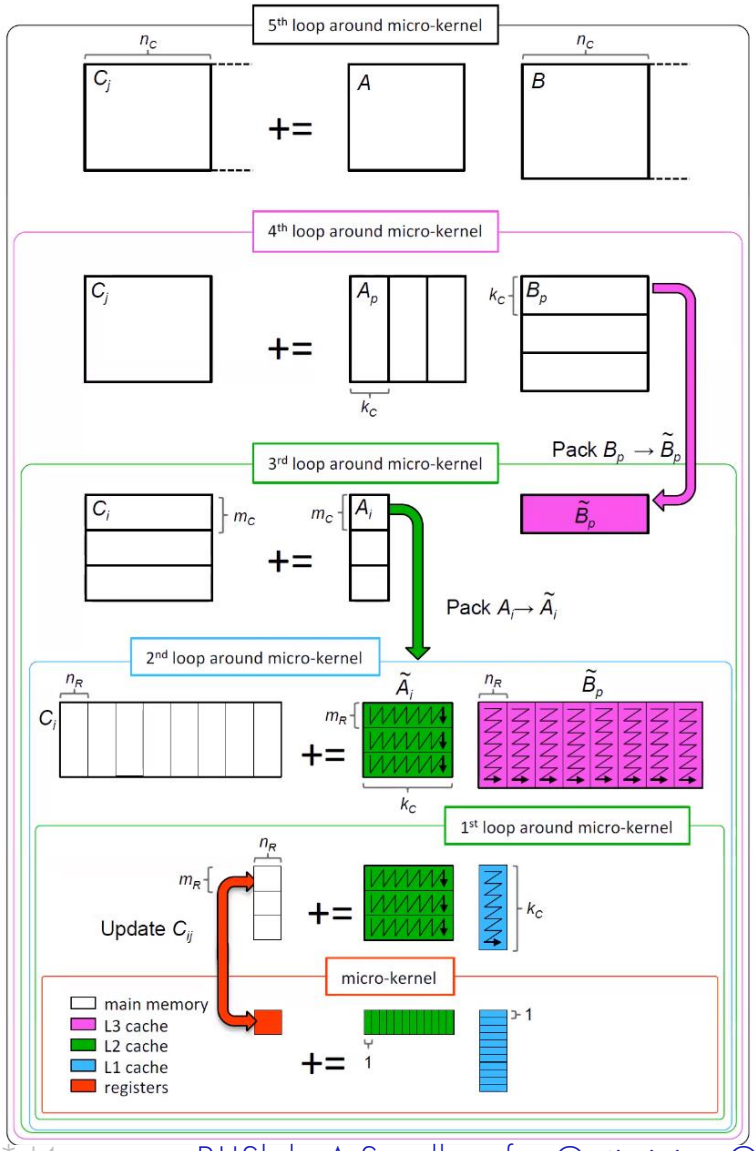



```
Loop 5  for jc = 0 : n−1 steps of nc
          Jc = jc : jc+nc−1
Loop 4    for pc = 0 : k−1 steps of kc
            Pc = pc : pc+kc−1
            B(Pc, Jc) → Bc              // Pack into Bc
Loop 3      for ic = 0 : m−1 steps of mc
              Ic = ic : ic+mc−1
              A(Ic, Pc) → Ac            // Pack into Ac
            // Macro-kernel
Loop 2      for jr = 0 : nc−1 steps of nr
              Jr = jr : jr+nr−1
Loop 1        for ir = 0 : mc−1 steps of mr
                Ir = ir : ir+mr−1
              // Micro-kernel
Loop 0        for kr = 0 : kc−1
                Cc(Ir, Jr)
                  += Ac(Ir, kr)  Bc(kr, Jr)
              endfor
            endfor
          endfor
        endfor
```

- Этот подход используется в библиотеках OpenBLAS и BLIS.
- В **OpenBLAS** оптимизация начинается с **Loop 2**.

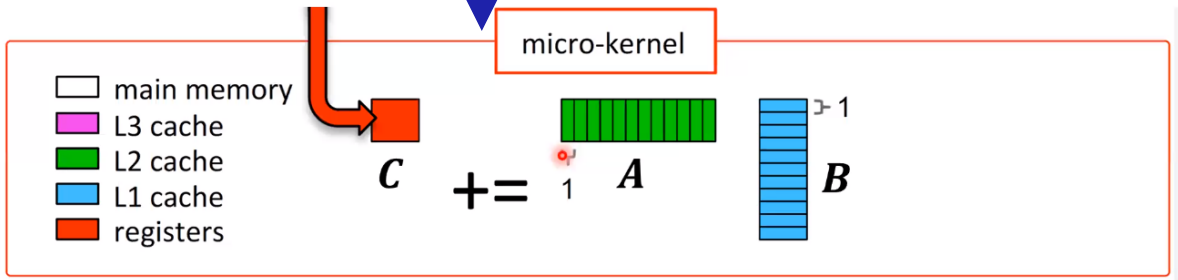
* Источник: BLISlab: A Sandbox for Optimizing GEMM

Goto,Geijn: Anatomy of High-Performance Matrix Multiplication



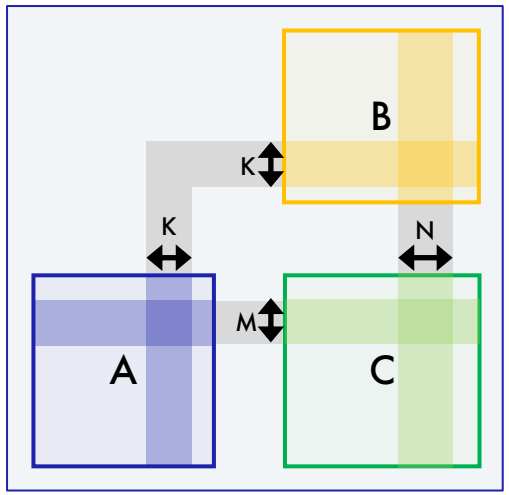
Loop 5 for $j_c = 0 : n-1$ steps of n_c
  $\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4 for $p_c = 0 : k-1$ steps of k_c
    $\mathcal{P}_c = p_c : p_c + k_c - 1$
    $B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow B_c$ // Pack into B_c
Loop 3 for $i_c = 0 : m-1$ steps of m_c
      $\mathcal{I}_c = i_c : i_c + m_c - 1$
      $A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow A_c$ // Pack into A_c
      // Macro-kernel
Loop 2 for $j_r = 0 : n_c - 1$ steps of n_r
        $\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1 for $i_r = 0 : m_c - 1$ steps of m_r
          $\mathcal{I}_r = i_r : i_r + m_r - 1$
          // Micro-kernel
Loop 0 for $k_r = 0 : k_c - 1$
            $C_c(\mathcal{I}_r, \mathcal{J}_r)$
              $+= A_c(\mathcal{I}_r, k_r)\ B_c(k_r, \mathcal{J}_r)$
          endfor
        endfor
      endfor
    endfor
  endfor
endfor

- Этот подход используется в библиотеках OpenBLAS и BLIS.
- В **OpenBLAS** оптимизация начинается с **Loop 2**.
- В **BLIS** оптимизация начинается с **Loop 0**, т.к. он может использоваться как часть имплементации нескольких разных алгоритмов библиотеки, что делает ее более гибкой для дальнейшего масштабирования.



* Источник: BLISlab: A Sandbox for Optimizing GEMM

Как ускоряться без ускорителей, но с расширениями?



Например, рассмотрим **умножение матриц**

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}.$$

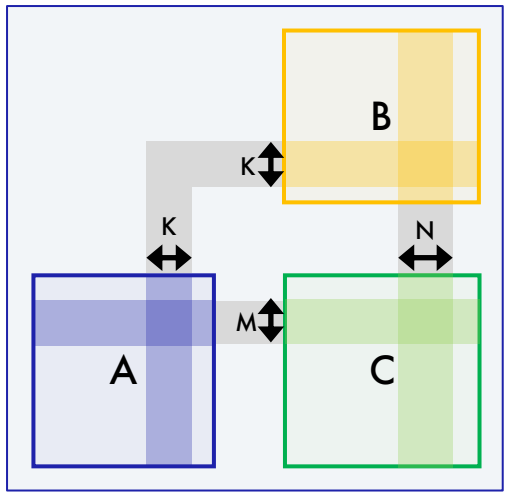
Алгоритм аккумуляции произведения очень прост:

$$C_{M\times N} = [0]_{M\times N}, \qquad \text{for i} = [0, M)$$
$$\text{for j} = [0, N)$$
$$\text{for k} = [0, \text{K}) \;\; C_i^j \mathrel{+}= A_i^k B_k^j.$$

Это $M \cdot N \cdot K$ умножений-сложений. **И пусть M, N, K такие что, все в кэше...**

Число операций не сократить...

Как ускоряться без ускорителей, но с расширениями?



Например, рассмотрим **умножение матриц**

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}.$$

Алгоритм аккумуляции произведения очень прост:

$$C_{M\times N} = [0]_{M\times N}, \qquad \text{for i} = [0, M)$$
$$\text{for j} = [0, N)$$
$$\text{for k} = [0, \text{K}) \;\; C_i^j \mathrel{+}= A_i^k B_k^j.$$

Это $M \cdot N \cdot K$ умножений-сложений. **И пусть M, N, K такие что, все в кэше...**

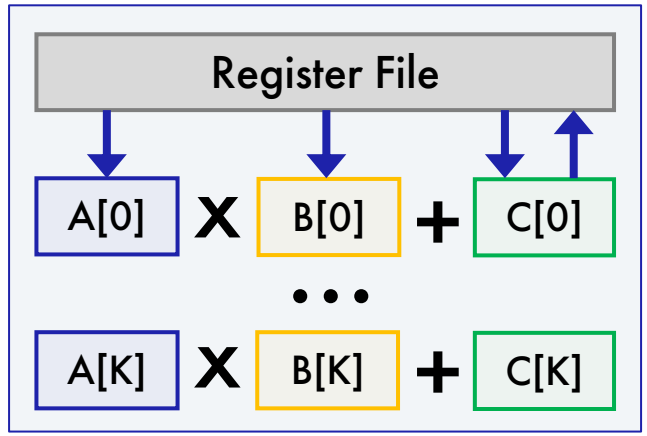
Число операций не сократить...

Но можно распараллелить!

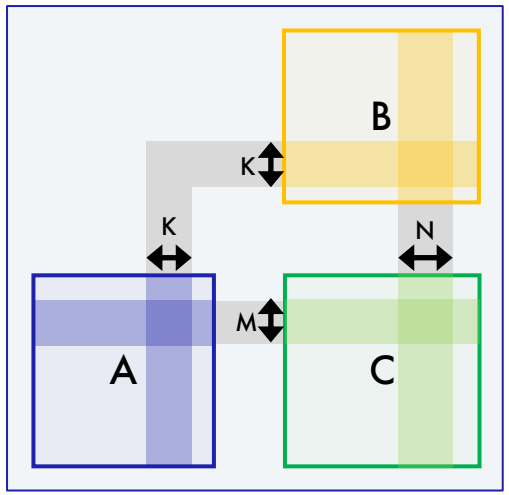
Например, векторизовать:

- загрузили $2K$ элементов,
- сделали K операций

} и так $M \cdot N$ раз.



Как ускоряться без ускорителей, но с расширениями?



Например, рассмотрим **умножение матриц**

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}.$$

Алгоритм аккумуляции произведения очень прост:

$$C_{M\times N} = [0]_{M\times N}, \qquad \text{for i} = [0, M)$$
$$\text{for j} = [0, N)$$
$$\text{for k} = [0, \text{K}) \;\; C_i^j \mathrel{+}= A_i^k B_k^j.$$

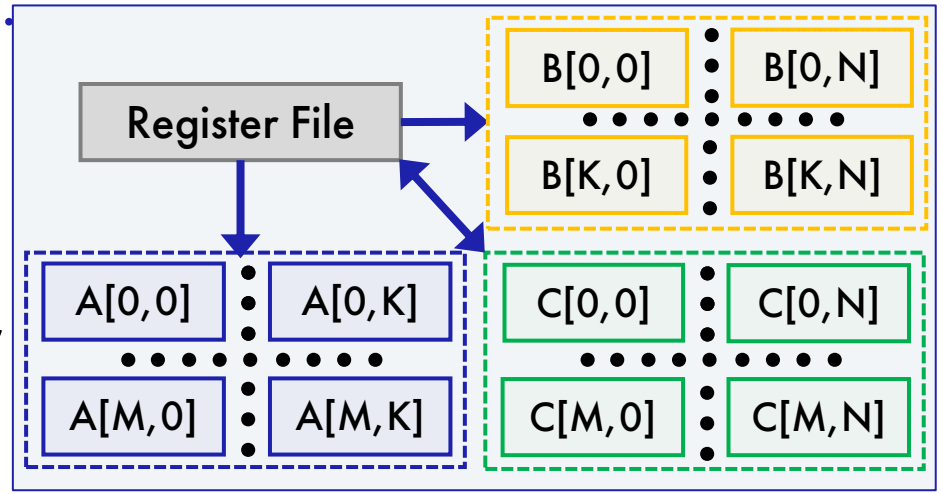
Это $M \cdot N \cdot K$ умножений-сложений. **И пусть M, N, K такие что, все в кэше...**

Число операций не сократить...

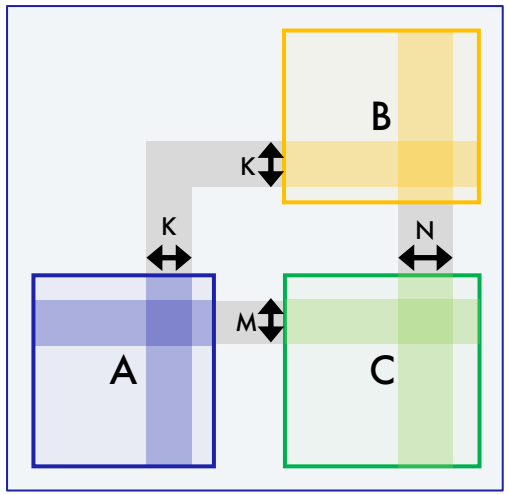
А можно сразу – плитками?

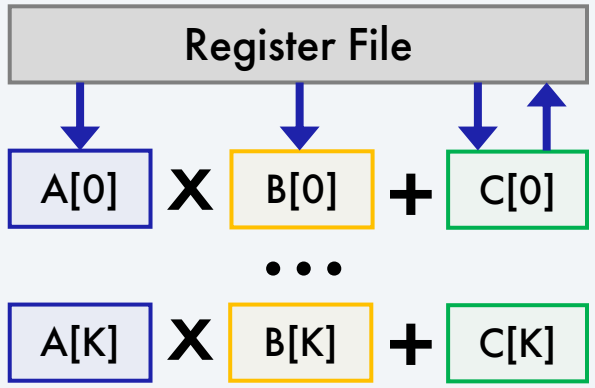
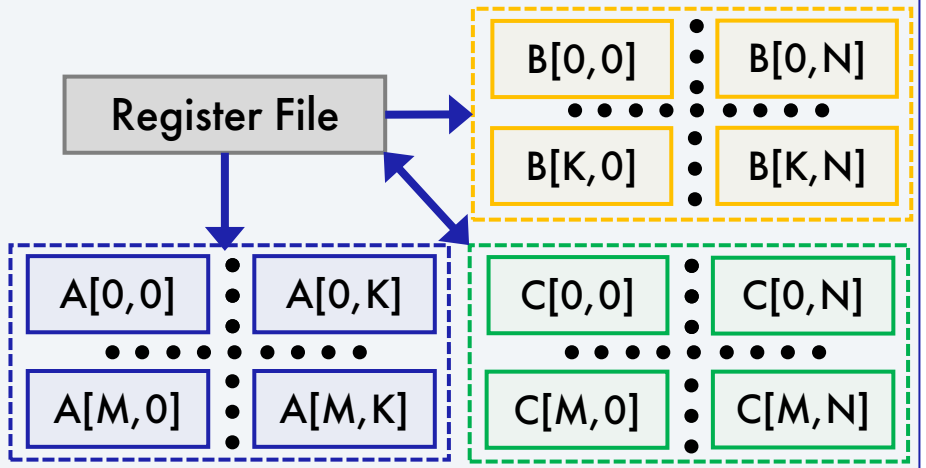
1 раз:

- загрузили $K \cdot (M + N)$ элементов,
- сделали $M \cdot N \cdot K$ операций

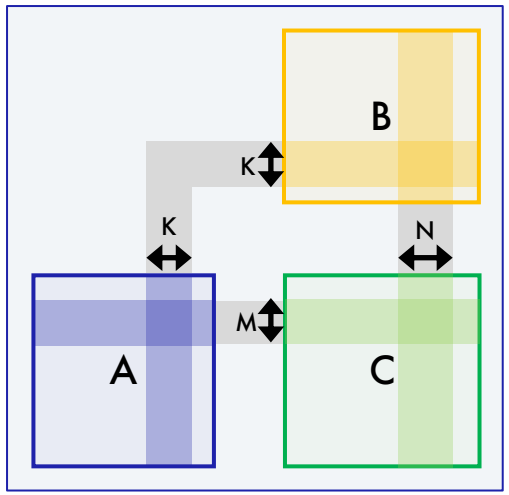


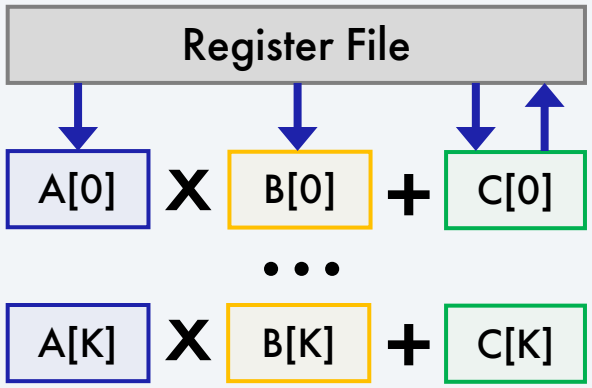
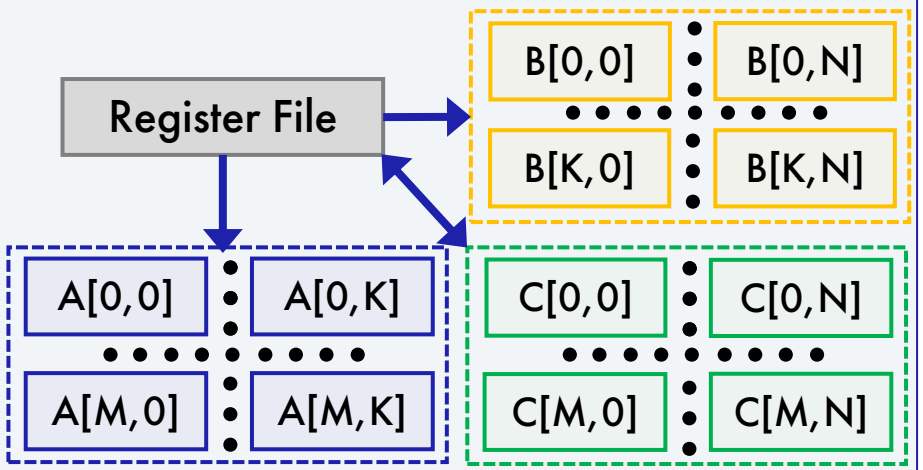
Какое расширение ускоряет сильнее?



Векторное расширение	VS	Матричное расширение
$M \cdot N$ раз: • загрузили $2K$ элементов, • сделали K операций.	>	1 раз: • загрузили $K \cdot (M + N)$ элементов, • сделали $M \cdot N \cdot K$ операций.
	<	**Операция «дороже»** (по тактам).
		

Какое расширение ускоряет сильнее?



Векторное расширение	VS	Матричное расширение
$M \cdot N$ раз: • загрузили $2K$ элементов, • сделали K операций.	>	1 раз: • загрузили $K \cdot (M + N)$ элементов, • сделали $M \cdot N \cdot K$ операций.
$\eta = \frac{1}{2} = O(1)$	<	$\eta = \frac{M \cdot N}{M + N} = \lvert M \sim N \rvert = O(N)$
	<	**Операция «дороже»** (по тактам).
		

Вычислительная интенсивность

$$\eta = \frac{\text{число операций}}{\text{число элементов}}$$

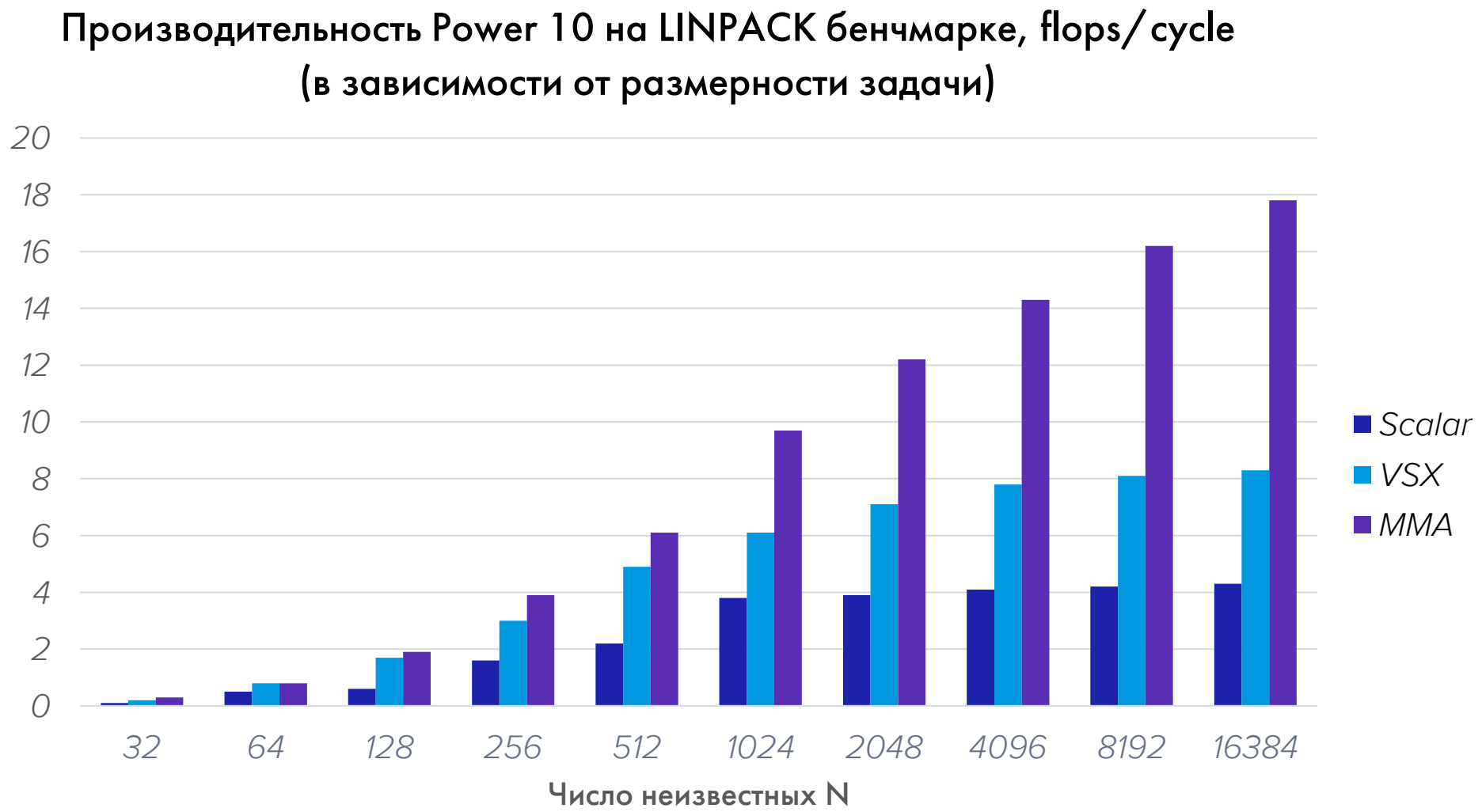
Что говорят открытые источники о производительности?

Расширение	Кейс	Ускорение, раз	vs Vector
Intel AMX	ResNet50 (batch size = 16), FP32	**2,2**	Intel VNNI
Apple AMX	Задачи AR/VR, ML, CV	**2**	Neon
Power MMA	GEMM, FP64 ($N \times 128$, N кратно 128)	**2,6**	VSX
	LINPACK (N = 16384)	**2,2**	
SiFive VCIX	GEMM, INT8	**12**	RVV
	MobileNet v1(batch size = 1),VLEN = 512 бит	**6**	
Arm SME	GEMM, FP32 ($M = N = K = 32..256$), VL = 512 бит	**2,2–6,4**	SVE

*Для Arm SME сравнивалось число тактов (на симуляторе). Во всех остальных случаях – время счета.

* Источники: Intel AMX, Apple AMX, Power MMA, SiFive VCIX, Arm SME

Какие реализации лучше масштабируются?



* Источник: https://developer.ibm.com/tutorials/power10-business-inferencing-at-scale-with-mma/

Виды матричных расширений (по связи с векторным)

Интегрированные

SiFive Intelligence Extension

Гибридные

Power MMA

Arm SME

Независимые

Intel AMX

Apple AMX

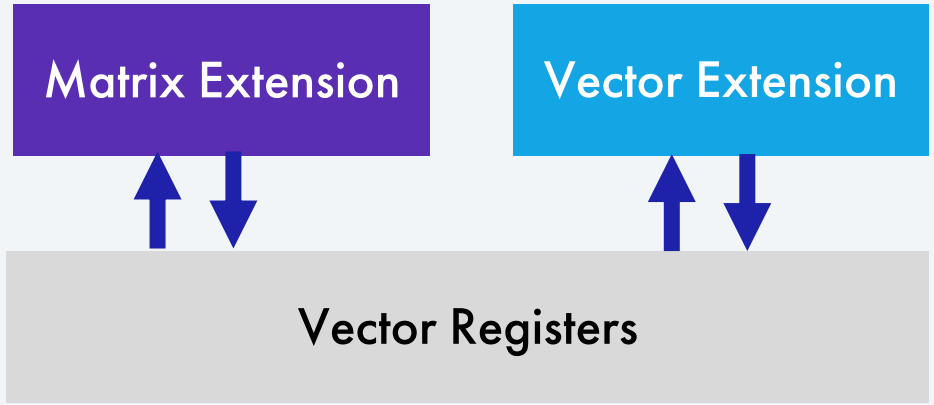
Виды матричных расширений (по связи с векторным)

Интегрированные
SiFive Intelligence Extension

Гибридные
Power MMA
Arm SME

Независимые
Intel AMX
Apple AMX

- Переиспользуют **векторные** регистры для блоков матриц-**сомножителей**.
- Переиспользуют **векторные** регистры для блоков матрицы-**аккумулятора**.



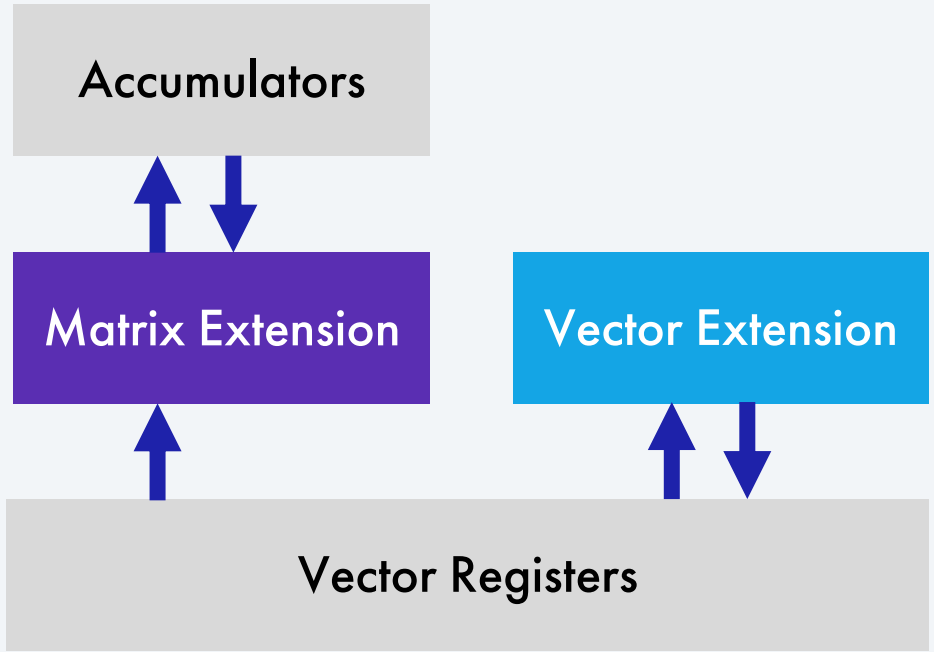
Виды матричных расширений (по связи с векторным)

Интегрированные
SiFive Intelligence Extension

Гибридные
Power MMA
Arm SME

Независимые
Intel AMX
Apple AMX

- Переиспользуют **векторные** регистры для блоков матриц-**сомножителей**.
- Вводят **независимые** регистры для блоков матрицы-**аккумулятора**.



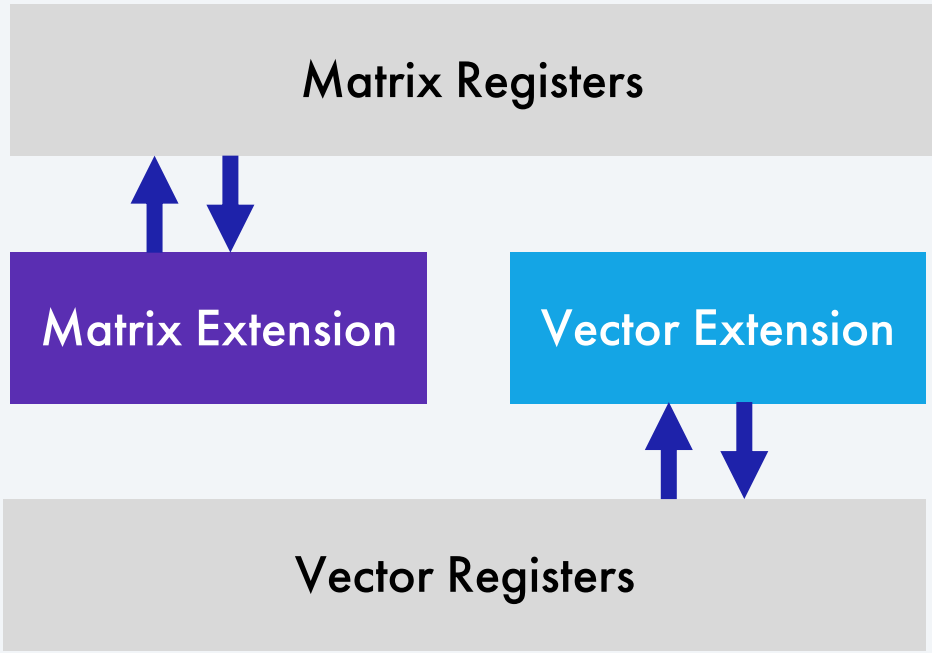
Виды матричных расширений (по связи с векторным)

Интегрированные
SiFive Intelligence Extension

Гибридные
Power MMA
Arm SME

Независимые
Intel AMX
Apple AMX

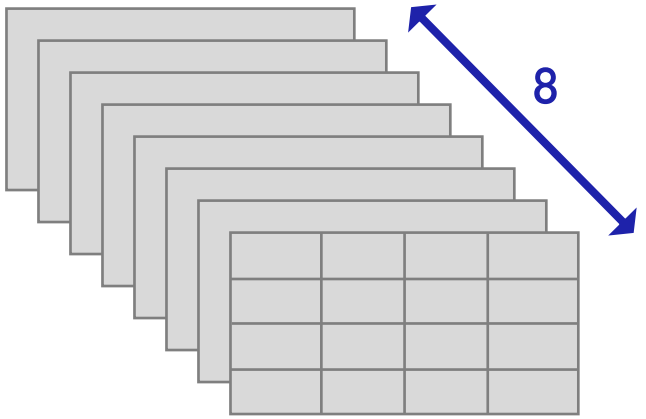
- Добавляют **матричные** регистры для блоков матриц-**сомножителей**.
- Добавляют **матричные** регистры для блоков матрицы-**аккумулятора**.



Intel AMX (Advanced Matrix Extension)

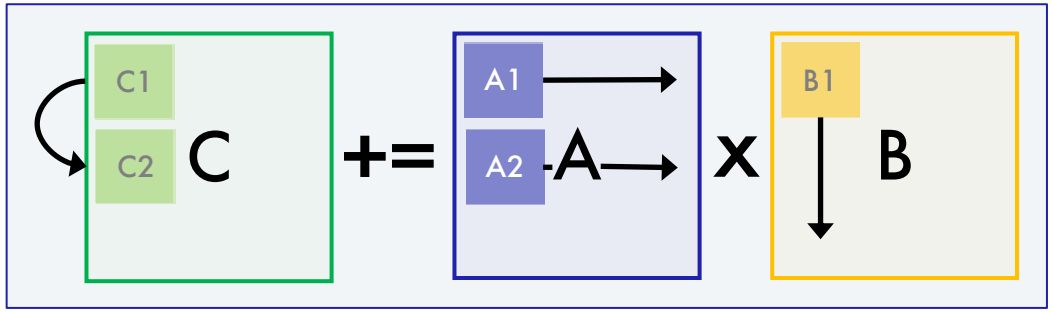
Тензорные регистры – «тайлы»:

- Новый расширяемый 2D регистровый файл.
- 8 регистров T0-T7 (2D), по 1Кб каждый.
- Базовые операции (load/store, clear, set и т.д.)



TMUL – инструкции матричного умножения:

- Использует 3 регистра: T2 += T1 * T0.
- Разбивает матрицы-операнды на «тайлы».



- Пока это единственная реализованная операция над «тайлами».

* Источник: https://www.nextplatform.com/2021/08/19/with-amx-intel-adds-ai-ml-sparkle-to-sapphire-rapids/

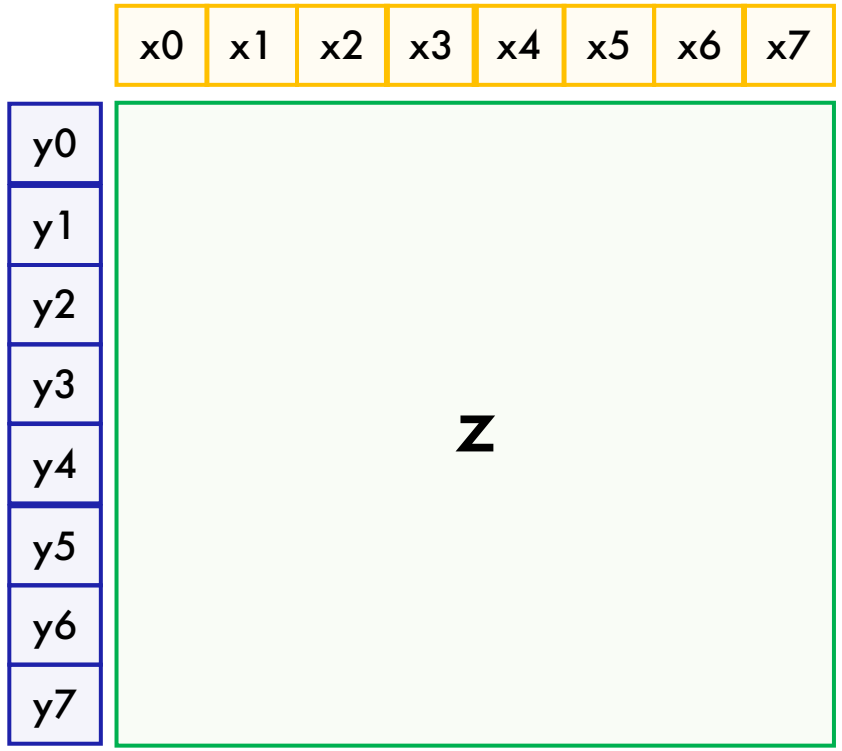
Apple AMX (Attached Matrix Extension)

Новое архитектурное пространство **5 Кб**, разбито на:

- **8 x регистров**,
- **8 y регистров**,

} каждый – 64-байтовый вектор;

- **z**, который может быть **сконфигурирован** как:
  - **1 регистр** 64 x 64 (32 x 32) по 8 (32) бита;
  - **2 регистра** 32 x 32 по 16 бит;
  - **4 регистра** 16 x 16 по 32 бита;
  - **8 регистров** 8 x 8 по 64 бита;
  - **64 регистра**, каждый – 64-байтовый вектор-строка.



* Источник: https://www.corsix.org/content/contrasting-intel-amx-and-apple-amx

Arm SME (Scalable Matrix Extension)

- **Векторные** (SVE/SVE2) **регистры** Z и P **для исходных операндов.**
  - Вместо длины вектора VL используется SVL (Streaming Vector Length)
  - SVL = {128, 256, 512, 1024, 2048} бит.
- Один новый 2D **матричный регистр** ZA **для аккумулятора**.
  - SVL_B строк, SVL_B – размер SVL в байтах.
  - SVL_B /N столбцов, N = {1, 4} – байт/элемент.

Симулятор

Поддерживается несколько операций над матрицами.

- **SME:** умножение матриц.
- **SME2:** GEMV, нелинейные решатели, разреженные матрицы.

* Источник: https://www.dcs.warwick.ac.uk/pmbs/pmbs22/PMBS/talk10.pdf

Power MMA (Matrix-Multiply Assist)

- Это **гибридное** матричное расширение?

  Позиционируется так: VSR[0..31] полностью отданы MMA.

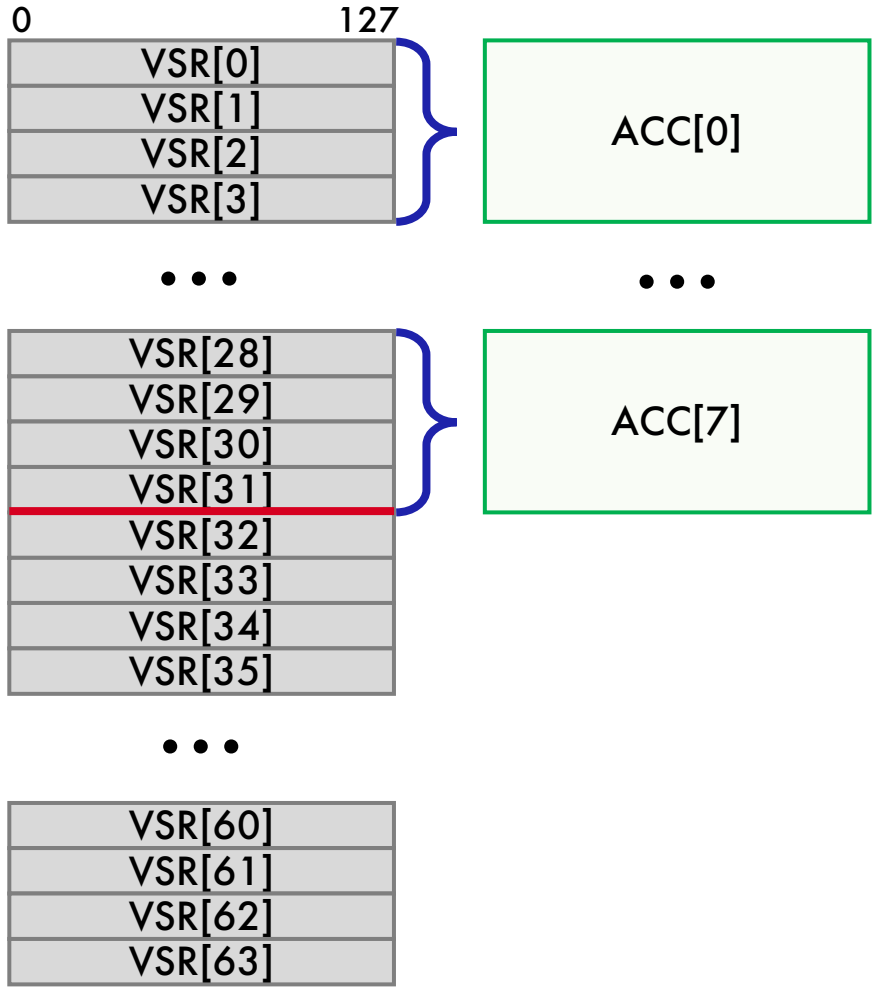
- Восемь **512-битных аккумуляторных регистра:** каждый из 4-х регистров векторного скалярного расширения (VSX).

- Поддержка в **OpenBLAS** (CORE=POWER10): оптимизации функций GEMM и TRMM для всех типов данных.

  Используется в NumPy, PyTorch и других фреймворках.

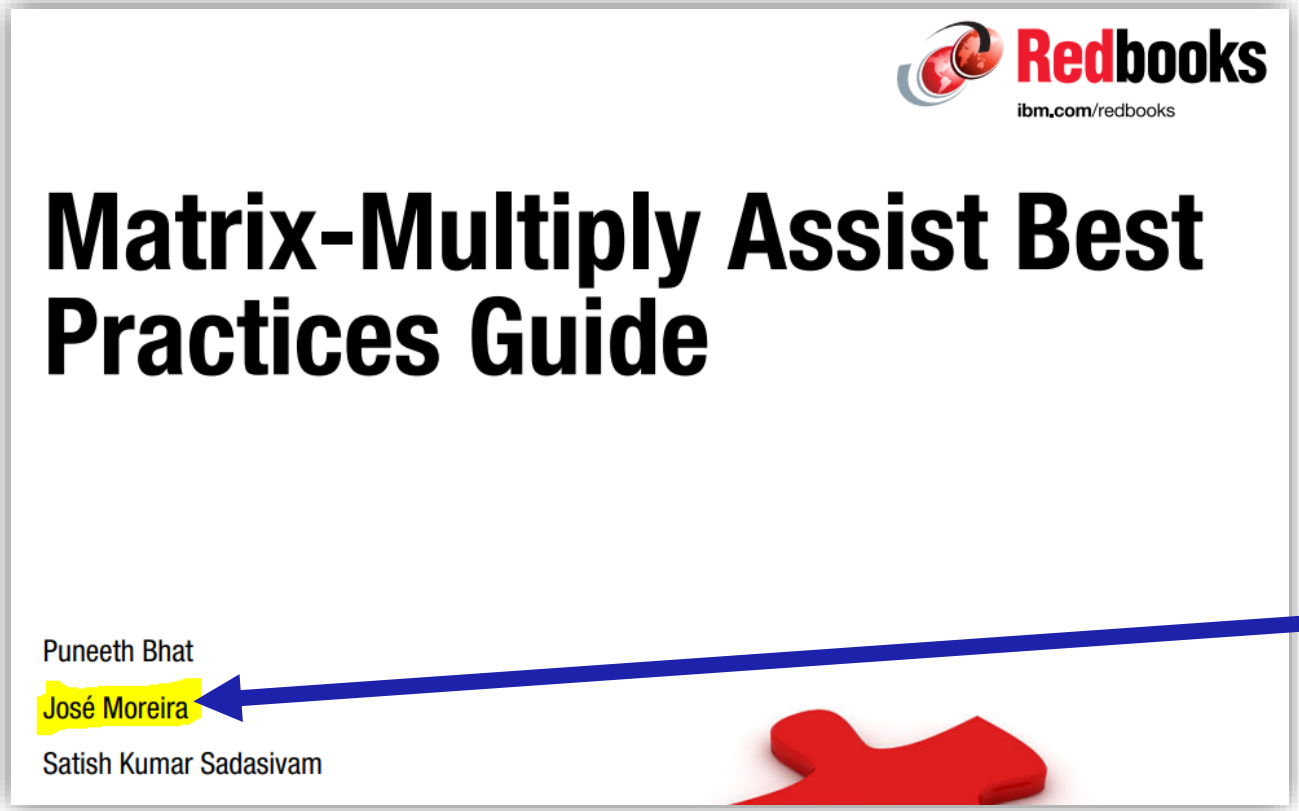
- Также имеются оптимизации в **Eigen**.

  Используется в OpenCV, TensorFlow и других фреймворках.

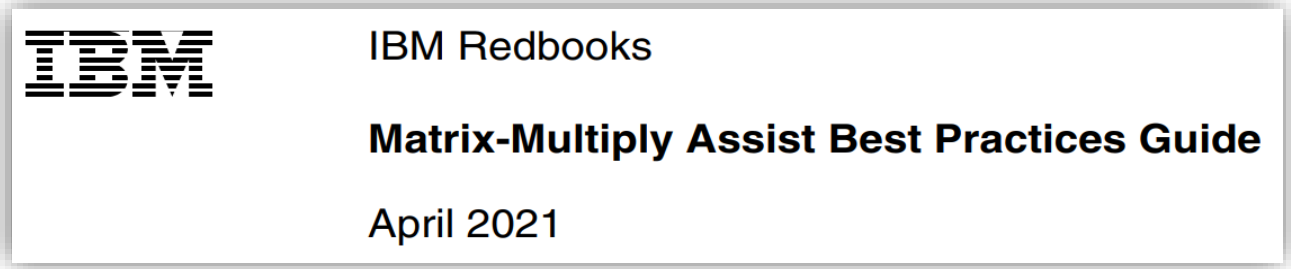


* Источник: https://developer.ibm.com/tutorials/power10-business-inferencing-at-scale-with-mma/

Power MMA: возможное влияние на RISC-V IME TG



- Председатель RISC-V Vector SIG.
- Инициировал создание RISC-V IME TG (Integrated Matrix Extension Task Group).
- Вице-председатель RISC-V IME TG.



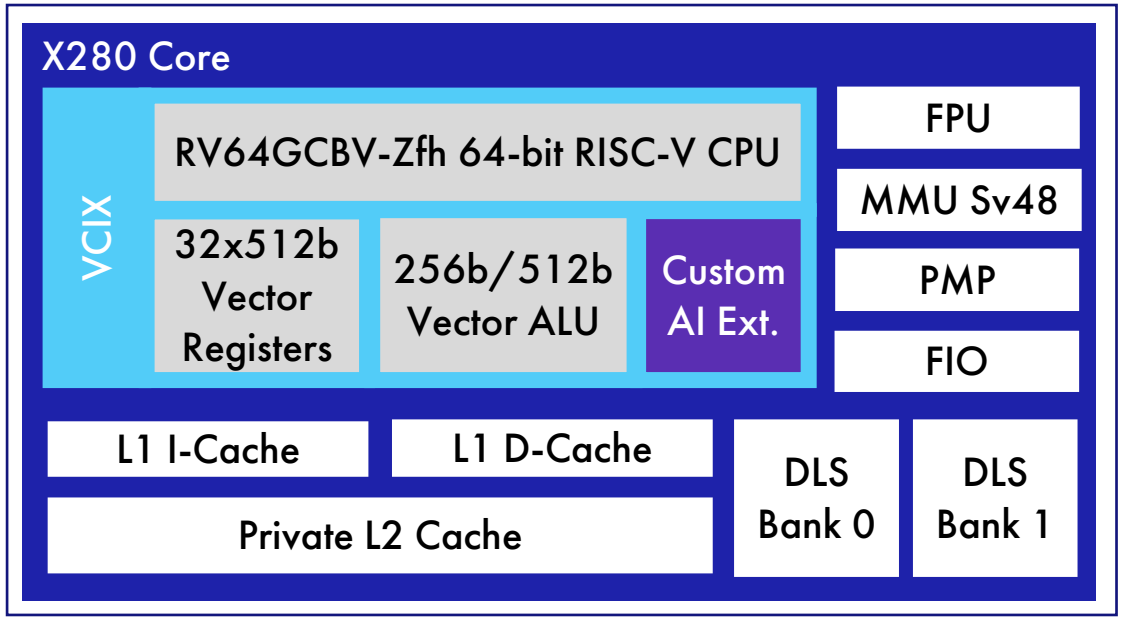
* Источник: https://www.redbooks.ibm.com/redpapers/pdfs/redp5612.pdf

SiFive VCIX (Vector Coprocessor Intelligence Extension)

Кастомное матричное расширение RISC-V, **интегрированное** в векторное.

- Появилось **в 2022 г.**
- 512-битные векторные регистры.

- Поддержка в **TensorFlow Lite:**
  - оптимизированы NN операции;
  - портированы многие NN модели.



* Источник: https://www.sifive.com/blog/sifive-intelligence-x280-as-ai-compute-host-google

SiFive Intelligence Extension: доступные спецификации

- SiFive **Int8** Matrix Multiplication Extensions Specification
  - **Xsfvqmaccqoq** Extension: Int8 Matrix Multiplication Instructions (4-by-8 and 8-by-4) + C intrinsics.
  - **Xsfvqmaccdod** Extension: Int8 Matrix Multiplication Instructions (2-by-8 and 8-by-2) + C intrinsics.

- Matrix Multiply Accumulate Instruction **(Bfloat16) Xsfvfwmaccqqq** Extension Specification + C intrinsics.

* Источник: https://www.sifive.com/documentation

Разработка стандартных матричных расширений RISC-V

Task Group (TG)		Integrated Matrix Extension (IME)	Attached Matrix Extension (AME)
Started on		11.12.2023	18.10.2023
Acting leadership		Guido Araujo (Unicamp) Jose Moreira (IBM)	Chou (Qiu) Jing (Alibaba) Philipp Tomsich (VRULL)
Proposed workplan			
Expected results	Specification	ISA + Intrinsics specification	
	Proof-of-concept implementations	• GNU binutils • GCC with intrinsics support • QEMU • LLVM MLIR • Linux kernel (e.g., context switching for matrix tiles) • oneDNN	+ • TensorFlow • PyTorch

* Источник: https://github.com/riscv-admin/vector

Определения и обстоятельства

Разработка T-Head RVM → RISC-V AME TG

Предложения RISC-V Vector SIG → RISC-V IME TG

Свежие вести с полей: Sparse Lives Matter

Заключение

T-Head: требования к матричному расширению RISC-V

- **Эффективность:**

  повысит производительность AI приложений за счет ускорения матричных операций.

- **Универсальность и масштабируемость:**

  эффективно работает **на широком спектре оборудования** – от маломощных периферийных устройств до высокопроизводительных серверов в дата-центрах.

- **Гибкость:**

  поддерживает широкий диапазон типов данных и размеров матриц.

- **Переносимость** двоичного кода (размеры регистров, аппаратная реализация).

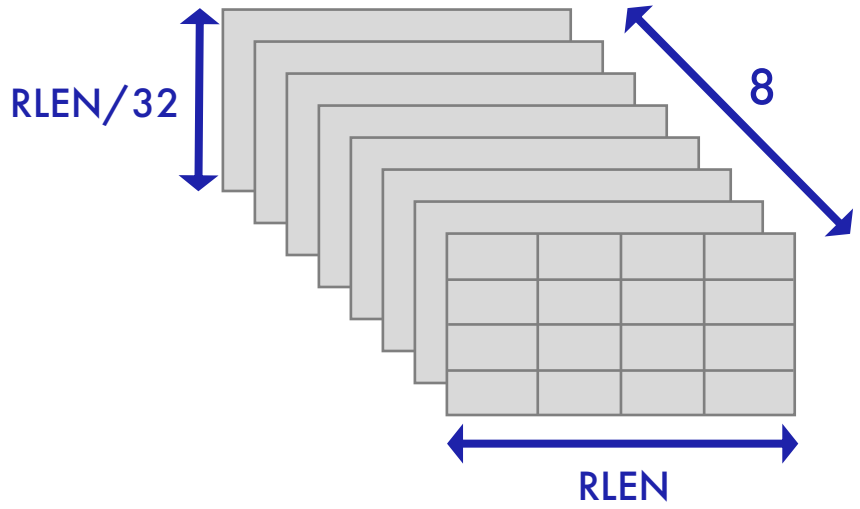
- **Перспективность:**

  легко расширить в будущем до полноценного **AI Matrix ++ Extension.**

* Источник: Matrix_Proposal_on_RISC-V_China_Summit_2023.pdf

Матричные регистры

- Восемь 2D **матричных регистров** для сомножителей и аккумуляторов.
  - Длина строки RLEN = 2^{n+6} = {128, 256, 512, ...} бит.
  - Число строк RLEN / 32 = 2^{n+1}.
  - Размер регистра MLEN = $RLEN^2$ / 32 = 2^{2n+7} бит.
  - Ширина элемента N = {4, 8, 16, 32, 64} бит.
  - Число столбцов RLEN / N.



* Источник: RISC-V_Matrix_Extensions_Proposal_202307.pdf

T-Head RVM ISA: всего 20+ инструкций

Группа инструкций	Количество инструкций	Краткое описание
Matrix MACC	3	Умножение блоков матриц и аккумуляция
Matrix operations	5	Прочие операции над матрицами и их частями: сложение/вычитание, векторное произведение векторов, сдвиг окна, выделение подматрицы. **Число поддерживаемых операций над матрицами будет увеличиваться.**
Memory access	4	Загрузка/выгрузка данных из матричных регистров
Move	6	Перемещение данных между регистрами (матричными или скалярными и матричными)
Matrix config	2	Конфигурирование матричных плиток
Others	2	Освобождение/обнуление матричных регистров

* Источник: Matrix_Proposal_on_RISC-V_China_Summit_2023.pdf

T-Head RVM Specification

- https://github.com/T-head-Semi/riscv-matrix-extension-spec
- **Call convention for RVM:** https://github.com/T-head-Semi/riscv-matrix-extension-spec/blob/master/doc/abi/riscv-cc.adoc
- **Programmer's Model:** https://github.com/T-head-Semi/riscv-matrix-extension-spec/blob/master/spec/matrix_body.adoc
- **RVM Intrinsic API:** https://github.com/T-head-Semi/riscv-matrix-extension-spec/blob/master/doc/intrinsic/rvm-intrinsic-api.adoc
- **Предлагаемая нумерация регистров:** https://github.com/T-head-Semi/riscv-matrix-extension-spec/blob/master/doc/abi/riscv-dwarf.adoc
- **Demo:** https://github.com/T-head-Semi/riscv-matrix-extension-spec/tree/master/demos
- https://riscv.org/blog/2023/02/xuantie-matrix-multiply-extension-instructions/

T-Head RVM Demo

- Тулчейн (XuanTie QEMU)
- **Система логирования:** общая статистика, детализированная статистика по инструкциям, вызовам функций и т.д.
- **RVV и RVM реализации** для:
  - GEMM ($M = N = K = 160$);
  - ResNet50.

  }
  - INT8;
  - FP16.
- Число инструкций **сокращается в 2-14 раз по сравнению с RVV.**

```
# cpf stat: gemm/gemm_fp16.elf
#
# ............/gemm_fp16.log .

    total cycles:          2151845
    total instructions:    116361
    CPI:                   18.493
    Read bytes:            2067351
    Write bytes:           62150

  [ icache ]
    icache refs:       118137

[ instruction summary ]
  load     instructions:    2671
  store    instructions:    1339
  abs jump instructions:    523
  branch   instructions:    12219
  call     instructions:    180
  dsp      instructions:    0
  vdsp     instructions:    24
  matrix   instructions:    50840
  float    instructions:    12
  rts      instructions:    0
  rte      instructions:    0
  alu      instructions:    47190
```

* Источник: https://github.com/T-head-Semi/riscv-matrix-extension-spec/tree/master/demos

Call for Matrix TG

Attached Matrix TG
Понедельник, 29 января · 6:00–6:55PM
Кажд. 2 нед. – понедельник

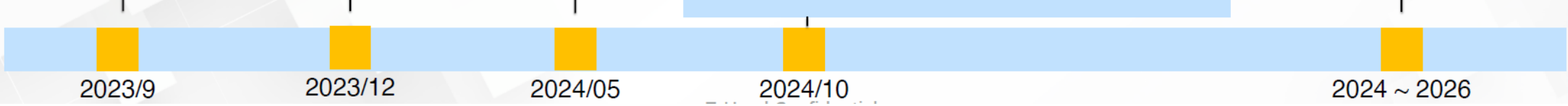
pre-inception stage
- Invite experts and engineers join the project
- Arrange meetings to discuss about matrix extension and AI applications
- Call for AI matrix extension spec proposals
- Gap analysis for matrix extension

milestone1
- Release matrix extension spec v0.1
- Choose benchmark to value all the proposals

milestone2
- Release AI matrix extension spec v0.5
- Toolchain (assembler/complier etc.) ready
- Simulator/Emulator ready

milestone3
- Ratify AI matrix extension spec v1.0
- Matrix compatibility test suit
- AI compiler stack and library ready

future plan
- Cooperate with other RISCV group
- OS supported(Linux/Android)
- AI framework supported(TensorFlow/PyTorch)
- Chip released including matrix extension
- Matrix++ extension



* Источник: RISC-V_Matrix_Extensions_Proposal_202307.pdf

Определения и обстоятельства

Разработка T-Head RVM ⟶ RISC-V AME TG

Предложения RISC-V Vector SIG ⟶ RISC-V IME TG

Свежие вести с полей: Sparse Lives Matter

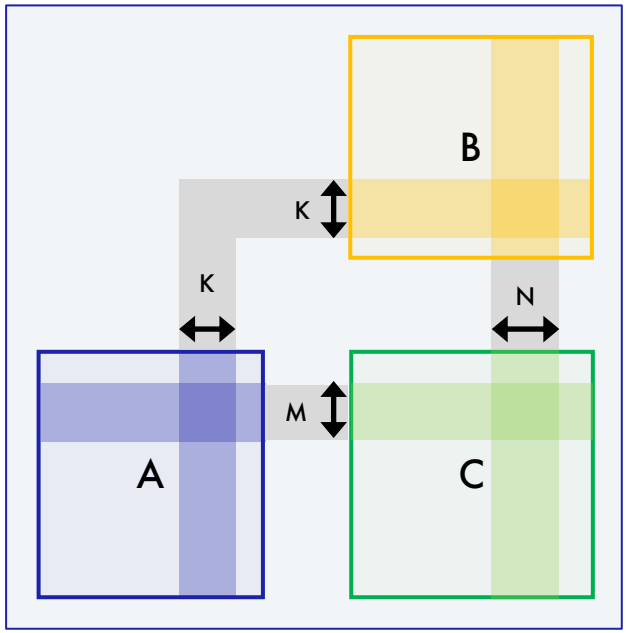
Заключение

RISC-V Vector SIG

Chair: Jose Moreira (IBM)
Vice-chair: Abel Bernabeu (Esperanto)

- Группа, ответственная за рассмотрение всех векторных расширений RISC-V ISA в будущем.
- Оценивает новые предложения, при необходимости создает свои собственные и работает над созданием целевых групп (TG) для новых расширений.
- https://github.com/riscv-admin/vector
- **Meeting minutes:** https://github.com/riscv-admin/vector/tree/main/minutes
- С сентября 2023 г. рассматривали возможные варианты **стандартного интегрированного матричного расширения** (катализатор – действия T-Head по созданию целевой группы для разработки стандартного независимого матричного расширения).
- **11 декабря 2023 г.** сформировали RISC-V IME TG (Integrated Matrix Facility Task Group).

Общие предположения



Рассматривается умножение блоков матриц

$$C_{M\times N} = A_{M\times K} \times B_{K\times N}.$$

Первый сомножитель рассматриваем как **вектор** из K **столбцов**:

$$A_{M\times K} = [A^0 \quad \cdots \quad A^{K-1}].$$

Второй сомножитель рассматриваем как **вектор** из K **строк**:

$$B_{K\times N} = [B_0 \quad \cdots \quad B_{K-1}]^T.$$

Тогда для умножения блоков можно использовать алгоритм

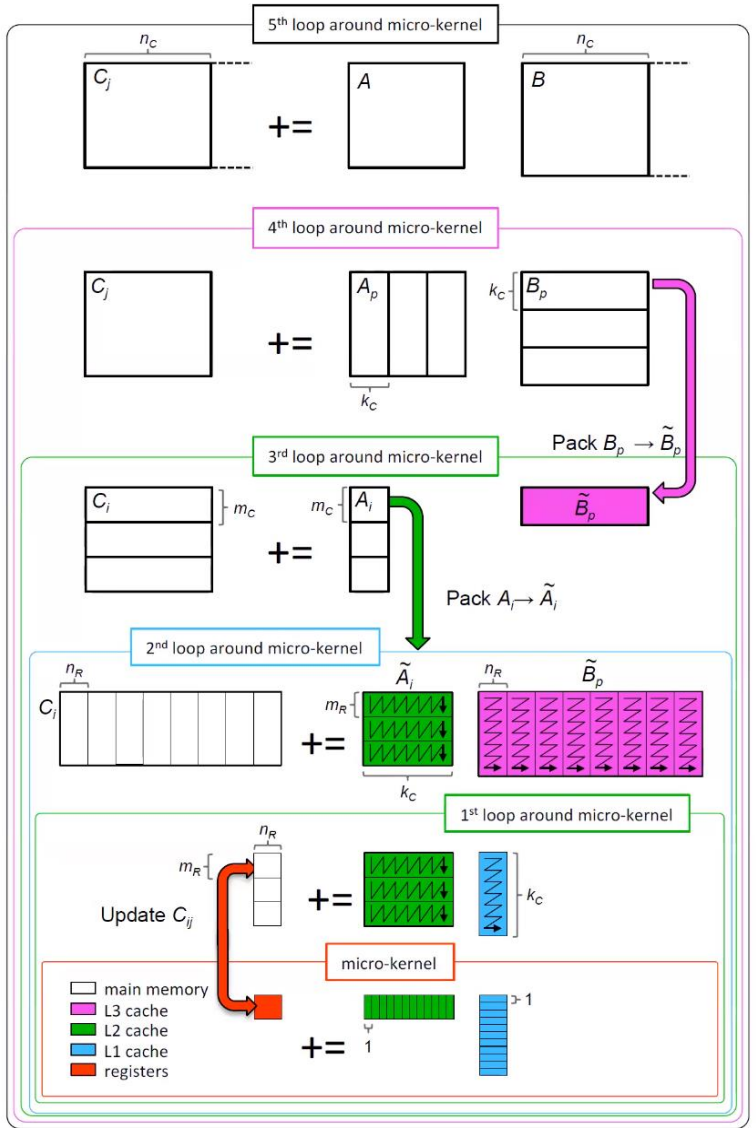
$$C_{M\times N} = [0]_{M\times N}, \qquad \text{for k} = [0, \text{K}) \quad C_{M\times N} \mathrel{+}= A^k \times B_k.$$

Таким образом, рассматриваем **внешнее произведение** векторов:

$$A^k \times B_k = \begin{bmatrix} A_0^k \\ \vdots \\ A_{M-1}^k \end{bmatrix} \times [B_k^0 \quad \cdots \quad B_k^{N-1}] = \begin{bmatrix} A_0^k B_k^0 & \cdots & A_0^k B_k^{N-1} \\ \vdots & \ddots & \vdots \\ A_{M-1}^k B_k^0 & \cdots & A_{M-1}^k B_k^{N-1} \end{bmatrix}.$$

* Источник: https://github.com/riscv-admin/vector

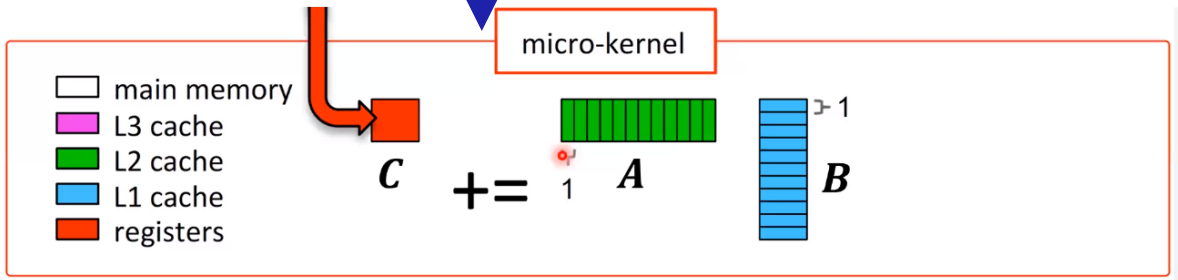
Определение размеров блоков



Loop 5 for $j_c = 0 : n-1$ steps of n_c
$\mathcal{J}_c = j_c : j_c + n_c - 1$
Loop 4 for $p_c = 0 : k-1$ steps of k_c
$\mathcal{P}_c = p_c : p_c + k_c - 1$
$B(\mathcal{P}_c, \mathcal{J}_c) \rightarrow B_c$ // Pack into B_c
Loop 3 for $i_c = 0 : m-1$ steps of m_c
$\mathcal{I}_c = i_c : i_c + m_c - 1$
$A(\mathcal{I}_c, \mathcal{P}_c) \rightarrow A_c$ // Pack into A_c
// Macro-kernel
Loop 2 for $j_r = 0 : n_c - 1$ steps of n_r
$\mathcal{J}_r = j_r : j_r + n_r - 1$
Loop 1 for $i_r = 0 : m_c - 1$ steps of m_r
$\mathcal{I}_r = i_r : i_r + m_r - 1$
// Micro-kernel
Loop 0 for $k_r = 0 : k_c - 1$
$C_c(\mathcal{I}_r, \mathcal{J}_r)$
$+= A_c(\mathcal{I}_r, k_r) \; B_c(k_r, \mathcal{J}_r)$
endfor
endfor
endfor
endfor
endfor

Goto, Geijn: Anatomy of High-Performance Matrix Multiplication

- Рассматривается уровень micro-kernel, как в библиотеке **BLIS**.
- Может быть частью имплементации разных алгоритмов.



* Источник: BLISlab: A Sandbox for Optimizing GEMM

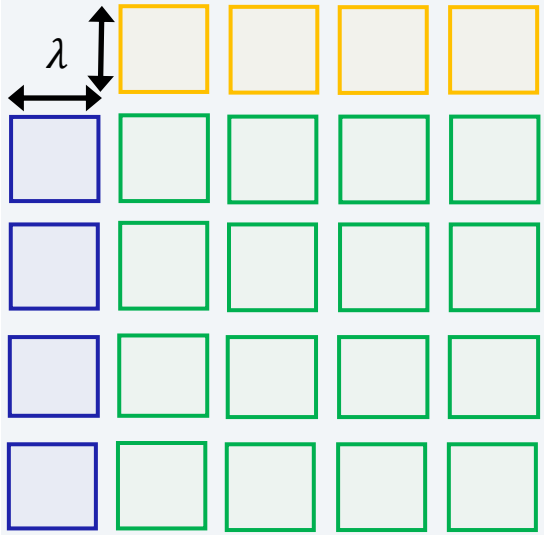
Общие предположения о регистрах

- Архитектурное пространство состоит из **32 векторных регистров**.
- Число элементов, помещающихся в один регистр, обозначим как L .
- Векторные регистры используются как для блоков двух матриц-сомножителей, так и для блоков матрицы-аккумулятора.
- Под блоки матрицы-аккумулятора отводится **16 векторных регистров**. ← **Половина, как в Power MMA**
- Число регистров, используемых для хранения элементов блоков матриц-сомножителей, обсуждается.
  - Определит, сколько элементов матриц A и B будет загружаться.
  - Вычислительная интенсивность $\eta = \frac{\text{число операций } mul{-}add}{\text{число загруженных элементов}}$ операций/элемент.

* Источник: https://github.com/riscv-admin/vector

Варианты декомпозиции операндов по регистрам

A: 16 + 4 + 4 регистра



- Все плитки матриц

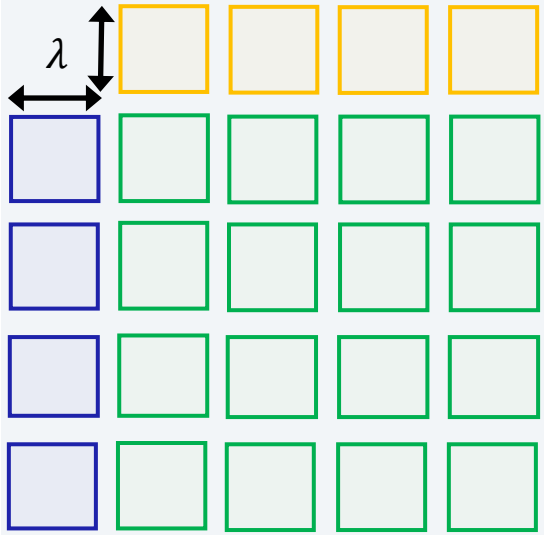
$$\lambda \times \lambda, \quad \lambda = \sqrt{L}.$$

- 1 плитка на регистр.

* Источник: https://github.com/riscv-admin/vector

Варианты декомпозиции операндов по регистрам

A: 16 + 4 + 4 регистра

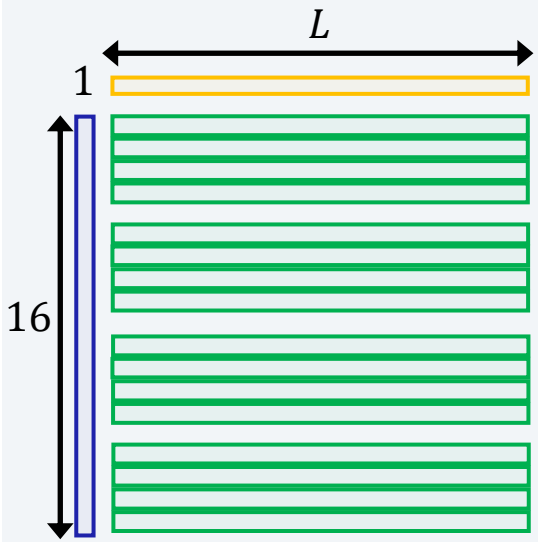


- Все плитки матриц

  $\lambda \times \lambda, \quad \lambda = \sqrt{L}.$

- 1 плитка на регистр.

B: 16 + ceil(16\L) + 1 регистр



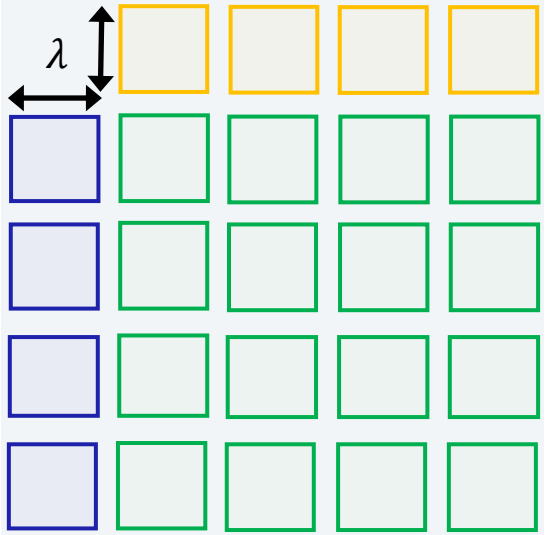
16 элементов столбца из A умножаем на L элементов строки из B.

Похоже на SiFive VCIX

* Источник: https://github.com/riscv-admin/vector

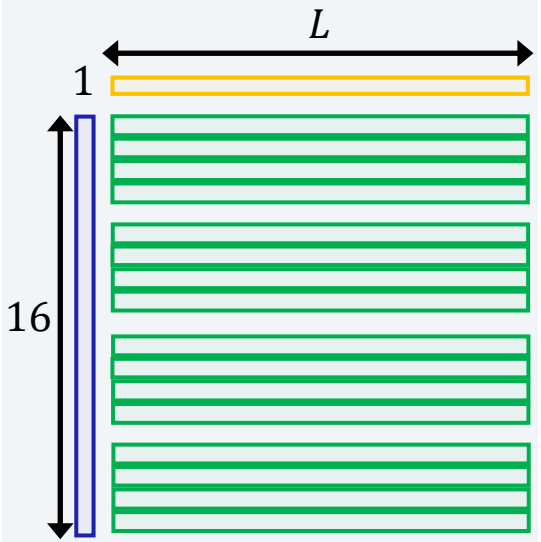
Варианты декомпозиции операндов по регистрам

A: 16 + 4 + 4 регистра



- Все плитки матриц $\lambda \times \lambda,\ \ \lambda = \sqrt{L}.$
- 1 плитка на регистр.

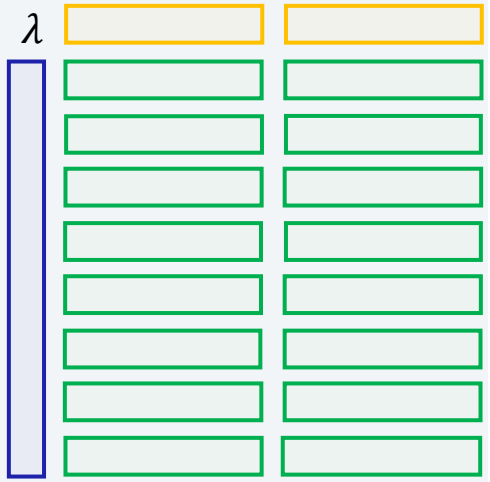
B: 16 + ceil(16\L) + 1 регистр



16 элементов столбца из A умножаем на L элементов строки из B.

Похоже на SiFive VCIX

C: 16 + ceil(8λ²/L) + 2 регистра

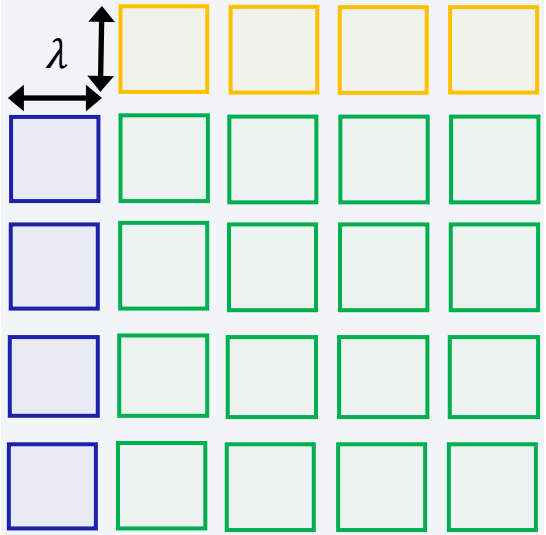


Полагаем, что элементы векторного регистра – матрицы $\lambda \times \lambda$.

* Источник: https://github.com/riscv-admin/vector

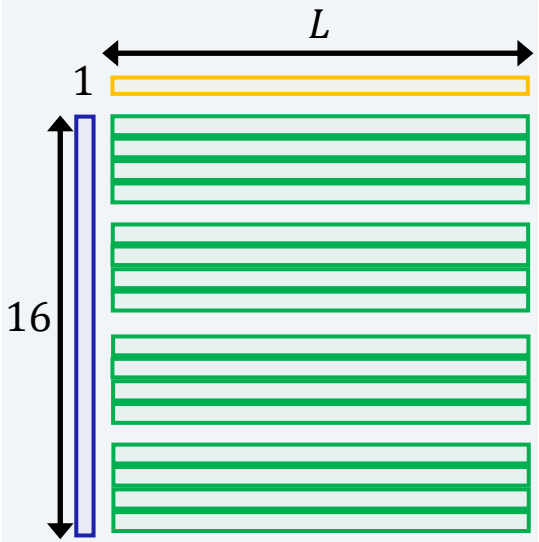
Варианты декомпозиции операндов по регистрам

A: 16 + 4 + 4 регистра



- Все плитки матриц $\lambda \times \lambda,\ \ \lambda = \sqrt{L}.$
- 1 плитка на регистр.

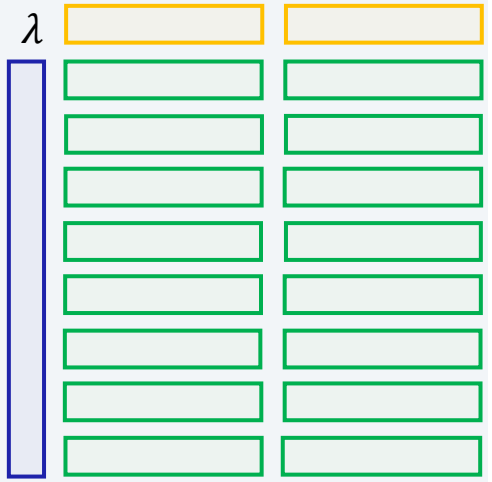
B: 16 + ceil(16\L) + 1 регистр



16 элементов столбца из A умножаем на L элементов строки из B.

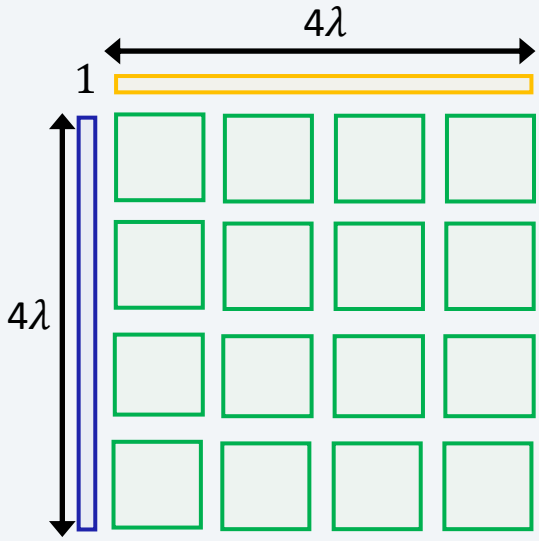
Похоже на SiFive VCIX

C: 16 + ceil(8λ²/L) + 2 регистра



Полагаем, что элементы векторного регистра – матрицы $\lambda \times \lambda$.

D: 16 + 2 * ceil(4/ λ) регистра

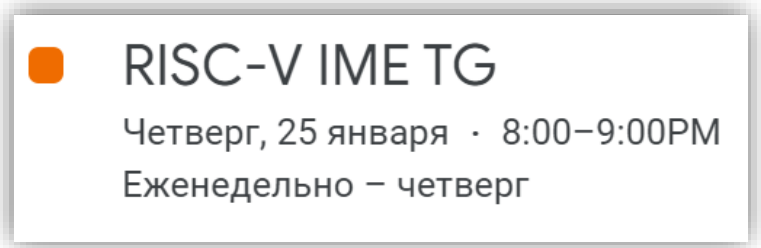


$4\lambda = 4\sqrt{L}$ элементов столбца A умножаем на 4λ элементов строки B.

* Источник: https://github.com/riscv-admin/vector

IME Task Group Proposal

- **HPC's GEMM as the use case for the extension.**
- Goal: ~90% peak performance for GEMM kernels; ~2x performance (vs vector) for other kernels.
- **Data types for HPC use cases** (32, 64-bit).
- **The added data types for AI/ML:** (4, 8, 16-bit).
- For a blue-print of the software kernel that we are accelerating we should look at the **BLIS microkernel**.
- The deliveries will be assumed :
  - specification of the operations,
  - instruction encoding,
  - compiler & simulation support,
  - library code development.



* Источник: https://github.com/riscv-admin/vector

Определения и обстоятельства

Разработка T-Head RVM ⟶ RISC-V AME TG

Предложения RISC-V Vector SIG ⟶ RISC-V IME TG

Свежие вести с полей: Sparse Lives Matter

Заключение

Vector SIG: первые обсуждения Sparse Matrix Extension

Нам следует включить разреженность в матричные расширения версии 2.0.
Нужно убедиться, что версия 1.0 не исключает этого.

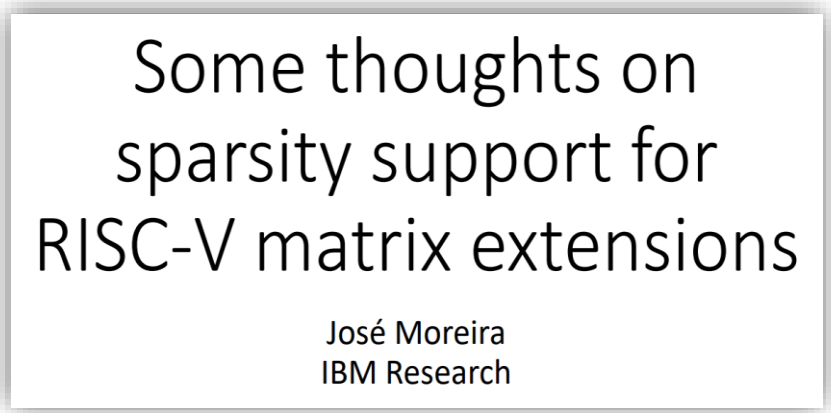
Разреженность — это сложная проблема,
и именно поэтому ее нужно начинать обсуждать как можно раньше.

Что еще более важно, сейчас самое время начать это обсуждение, если мы хотим перехватить работу над SparseBLAS.

Хосе Морейра,
председатель RISC-V Vector SIG

* Источник: https://github.com/riscv-admin/vector

Общие предположения



Рассматривается умножение блоков разреженной и плотной матриц

$$R_{M\times N} = S_{M\times K} \times D_{K\times N}.$$

Первый сомножитель – вектор из K **разреженных столбцов**:

$$S_{M\times K} = [S^0 \quad \cdots \quad S^{K-1}].$$

Разреженность (sparsity – % нулевых элементов) ≈ 50-90%

(характерно для матриц весов из DL моделей, для HPC – выше).

Второй сомножитель – вектор из K **плотных строк**:

$$D_{K\times N} = [D_0 \quad \cdots \quad D_{K-1}]^T.$$

Тогда для умножения блоков можно использовать алгоритм

$$R_{M\times N} = [0]_{M\times N}, \qquad \text{for k} = [0, \text{K}) \quad R_{M\times N} \mathrel{+}= S^k \times D_k.$$

Аккумулятор – плотная матрица.

* Источник: https://github.com/riscv-admin/vector

В чем отличие от произведения плотных матриц?

Если бы мы не знали, что S – разреженная.

$$S^k \times D_k = \begin{bmatrix} S_0^k \\ S_1^k \\ S_2^k \\ S_3^k \\ S_4^k \\ S_5^k \\ S_6^k \\ S_7^k \end{bmatrix} \times \begin{bmatrix} D_k^0 & D_k^1 & D_k^2 & D_k^3 \end{bmatrix} = \begin{bmatrix} S_0^k D_k^0 & S_0^k D_k^1 & S_0^k D_k^2 & S_0^k D_k^3 \\ S_1^k D_k^0 & S_1^k D_k^1 & S_1^k D_k^2 & S_1^k D_k^3 \\ S_2^k D_k^0 & S_2^k D_k^1 & S_2^k D_k^2 & S_2^k D_k^3 \\ S_3^k D_k^0 & S_3^k D_k^1 & S_3^k D_k^2 & S_3^k D_k^3 \\ S_4^k D_k^0 & S_4^k D_k^1 & S_4^k D_k^2 & S_4^k D_k^3 \\ S_5^k D_k^0 & S_5^k D_k^1 & S_5^k D_k^2 & S_5^k D_k^3 \\ S_6^k D_k^0 & S_6^k D_k^1 & S_6^k D_k^2 & S_6^k D_k^3 \\ S_7^k D_k^0 & S_7^k D_k^1 & S_7^k D_k^2 & S_7^k D_k^3 \end{bmatrix}$$

Для примера рассмотрим случай $M = 8,\ N = 4.$

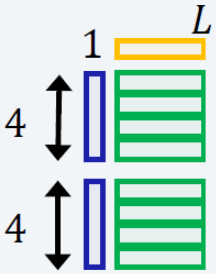
* Источник: https://github.com/riscv-admin/vector

В чем отличие от произведения плотных матриц?

Если бы мы не знали, что S – разреженная.

$$S^k \times D_k = \begin{bmatrix} S_0^k \\ S_1^k \\ S_2^k \\ S_3^k \\ S_4^k \\ S_5^k \\ S_6^k \\ S_7^k \end{bmatrix} \times \begin{bmatrix} D_k^0 & D_k^1 & D_k^2 & D_k^3 \end{bmatrix} = \begin{bmatrix} S_0^k D_k^0 & S_0^k D_k^1 & S_0^k D_k^2 & S_0^k D_k^3 \\ S_1^k D_k^0 & S_1^k D_k^1 & S_1^k D_k^2 & S_1^k D_k^3 \\ S_2^k D_k^0 & S_2^k D_k^1 & S_2^k D_k^2 & S_2^k D_k^3 \\ S_3^k D_k^0 & S_3^k D_k^1 & S_3^k D_k^2 & S_3^k D_k^3 \\ S_4^k D_k^0 & S_4^k D_k^1 & S_4^k D_k^2 & S_4^k D_k^3 \\ S_5^k D_k^0 & S_5^k D_k^1 & S_5^k D_k^2 & S_5^k D_k^3 \\ S_6^k D_k^0 & S_6^k D_k^1 & S_6^k D_k^2 & S_6^k D_k^3 \\ S_7^k D_k^0 & S_7^k D_k^1 & S_7^k D_k^2 & S_7^k D_k^3 \end{bmatrix}$$

Пусть $L = 4$ и аккумуляторные регистры имеют вид:



$$R = R + S^k \times D_k = \begin{bmatrix} R_0^0 & R_0^1 & R_0^2 & R_0^3 \\ R_1^0 & R_1^1 & R_1^2 & R_1^3 \\ R_2^0 & R_2^1 & R_2^2 & R_2^3 \\ R_3^0 & R_3^1 & R_3^2 & R_3^3 \\ R_4^0 & R_4^1 & R_4^2 & R_4^3 \\ R_5^0 & R_5^1 & R_5^2 & R_5^3 \\ R_6^0 & R_6^1 & R_6^2 & R_6^3 \\ R_7^0 & R_7^1 & R_7^2 & R_7^3 \end{bmatrix} + S^k \times D_k = \begin{bmatrix} R_0^0 + S_0^k D_k^0 & R_0^1 + S_0^k D_k^1 & R_0^2 + S_0^k D_k^2 & R_0^3 + S_0^k D_k^3 \\ R_1^0 + S_1^k D_k^0 & R_1^1 + S_1^k D_k^1 & R_1^2 + S_1^k D_k^2 & R_1^3 + S_1^k D_k^3 \\ R_2^0 + S_2^k D_k^0 & R_2^1 + S_2^k D_k^1 & R_2^2 + S_2^k D_k^2 & R_2^3 + S_2^k D_k^3 \\ R_3^0 + S_3^k D_k^0 & R_3^1 + S_3^k D_k^1 & R_3^2 + S_3^k D_k^2 & R_3^3 + S_3^k D_k^3 \\ R_4^0 + S_4^k D_k^0 & R_4^1 + S_4^k D_k^1 & R_4^2 + S_4^k D_k^2 & R_4^3 + S_4^k D_k^3 \\ R_5^0 + S_5^k D_k^0 & R_5^1 + S_5^k D_k^1 & R_5^2 + S_5^k D_k^2 & R_5^3 + S_5^k D_k^3 \\ R_6^0 + S_6^k D_k^0 & R_6^1 + S_6^k D_k^1 & R_6^2 + S_6^k D_k^2 & R_6^3 + S_6^k D_k^3 \\ R_7^0 + S_7^k D_k^0 & R_7^1 + S_7^k D_k^1 & R_7^2 + S_7^k D_k^2 & R_7^3 + S_7^k D_k^3 \end{bmatrix}$$

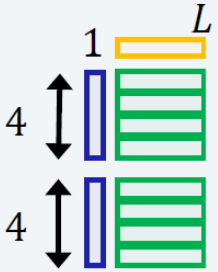
* Источник: https://github.com/riscv-admin/vector

В чем отличие от произведения плотных матриц?

Пусть sparsity равна 50% и портрет такой:

$$S^k \times D_k = \begin{bmatrix} S_0^k \\ 0 \\ S_2^k \\ S_3^k \\ 0 \\ 0 \\ S_6^k \\ 0 \end{bmatrix} \times \begin{bmatrix} D_k^0 & D_k^1 & D_k^2 & D_k^3 \end{bmatrix} = \begin{bmatrix} S_0^k D_k^0 & S_0^k D_k^1 & S_0^k D_k^2 & S_0^k D_k^3 \\ S_1^k D_k^0 & S_1^k D_k^1 & S_1^k D_k^2 & S_1^k D_k^3 \\ S_2^k D_k^0 & S_2^k D_k^1 & S_2^k D_k^2 & S_2^k D_k^3 \\ S_3^k D_k^0 & S_3^k D_k^1 & S_3^k D_k^2 & S_3^k D_k^3 \\ S_4^k D_k^0 & S_4^k D_k^1 & S_4^k D_k^2 & S_4^k D_k^3 \\ S_5^k D_k^0 & S_5^k D_k^1 & S_5^k D_k^2 & S_5^k D_k^3 \\ S_6^k D_k^0 & S_6^k D_k^1 & S_6^k D_k^2 & S_6^k D_k^3 \\ S_7^k D_k^0 & S_7^k D_k^1 & S_7^k D_k^2 & S_7^k D_k^3 \end{bmatrix}$$

Пусть $L = 4$ и аккумуляторные регистры имеют вид:



$$R = R + S^k \times D_k = \begin{bmatrix} R_0^0 & R_0^1 & R_0^2 & R_0^3 \\ R_1^0 & R_1^1 & R_1^2 & R_1^3 \\ R_2^0 & R_2^1 & R_2^2 & R_2^3 \\ R_3^0 & R_3^1 & R_3^2 & R_3^3 \\ R_4^0 & R_4^1 & R_4^2 & R_4^3 \\ R_5^0 & R_5^1 & R_5^2 & R_5^3 \\ R_6^0 & R_6^1 & R_6^2 & R_6^3 \\ R_7^0 & R_7^1 & R_7^2 & R_7^3 \end{bmatrix} + S^k \times D_k = \begin{bmatrix} R_0^0 + S_0^k D_k^0 & R_0^1 + S_0^k D_k^1 & R_0^2 + S_0^k D_k^2 & R_0^3 + S_0^k D_k^3 \\ R_1^0 + S_1^k D_k^0 & R_1^1 + S_1^k D_k^1 & R_1^2 + S_1^k D_k^2 & R_1^3 + S_1^k D_k^3 \\ R_2^0 + S_2^k D_k^0 & R_2^1 + S_2^k D_k^1 & R_2^2 + S_2^k D_k^2 & R_2^3 + S_2^k D_k^3 \\ R_3^0 + S_3^k D_k^0 & R_3^1 + S_3^k D_k^1 & R_3^2 + S_3^k D_k^2 & R_3^3 + S_3^k D_k^3 \\ R_4^0 + S_4^k D_k^0 & R_4^1 + S_4^k D_k^1 & R_4^2 + S_4^k D_k^2 & R_4^3 + S_4^k D_k^3 \\ R_5^0 + S_5^k D_k^0 & R_5^1 + S_5^k D_k^1 & R_5^2 + S_5^k D_k^2 & R_5^3 + S_5^k D_k^3 \\ R_6^0 + S_6^k D_k^0 & R_6^1 + S_6^k D_k^1 & R_6^2 + S_6^k D_k^2 & R_6^3 + S_6^k D_k^3 \\ R_7^0 + S_7^k D_k^0 & R_7^1 + S_7^k D_k^1 & R_7^2 + S_7^k D_k^2 & R_7^3 + S_7^k D_k^3 \end{bmatrix}$$

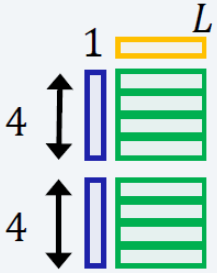
* Источник: https://github.com/riscv-admin/vector

В чем отличие от произведения плотных матриц?

Пусть sparsity равна 50% и портрет такой:

$$S^k \times D_k = \begin{bmatrix} S_0^k \\ 0 \\ S_2^k \\ S_3^k \\ 0 \\ 0 \\ S_6^k \\ 0 \end{bmatrix} \times \begin{bmatrix} D_k^0 & D_k^1 & D_k^2 & D_k^3 \end{bmatrix} = \begin{bmatrix} S_0^k D_k^0 & S_0^k D_k^1 & S_0^k D_k^2 & S_0^k D_k^3 \\ 0 & 0 & 0 & 0 \\ S_2^k D_k^0 & S_2^k D_k^1 & S_2^k D_k^2 & S_2^k D_k^3 \\ S_3^k D_k^0 & S_3^k D_k^1 & S_3^k D_k^2 & S_3^k D_k^3 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ S_6^k D_k^0 & S_6^k D_k^1 & S_6^k D_k^2 & S_6^k D_k^3 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Пусть $L = 4$ и аккумуляторные регистры имеют вид:



$$R = R + S^k \times D_k = \begin{bmatrix} R_0^0 & R_0^1 & R_0^2 & R_0^3 \\ R_1^0 & R_1^1 & R_1^2 & R_1^3 \\ R_2^0 & R_2^1 & R_2^2 & R_2^3 \\ R_3^0 & R_3^1 & R_3^2 & R_3^3 \\ R_4^0 & R_4^1 & R_4^2 & R_4^3 \\ R_5^0 & R_5^1 & R_5^2 & R_5^3 \\ R_6^0 & R_6^1 & R_6^2 & R_6^3 \\ R_7^0 & R_7^1 & R_7^2 & R_7^3 \end{bmatrix} + S^k \times D_k = \begin{bmatrix} R_0^0 + S_0^k D_k^0 & R_0^1 + S_0^k D_k^1 & R_0^2 + S_0^k D_k^2 & R_0^3 + S_0^k D_k^3 \\ R_1^0 + S_1^k D_k^0 & R_1^1 + S_1^k D_k^1 & R_1^2 + S_1^k D_k^2 & R_1^3 + S_1^k D_k^3 \\ R_2^0 + S_2^k D_k^0 & R_2^1 + S_2^k D_k^1 & R_2^2 + S_2^k D_k^2 & R_2^3 + S_2^k D_k^3 \\ R_3^0 + S_3^k D_k^0 & R_3^1 + S_3^k D_k^1 & R_3^2 + S_3^k D_k^2 & R_3^3 + S_3^k D_k^3 \\ R_4^0 + S_4^k D_k^0 & R_4^1 + S_4^k D_k^1 & R_4^2 + S_4^k D_k^2 & R_4^3 + S_4^k D_k^3 \\ R_5^0 + S_5^k D_k^0 & R_5^1 + S_5^k D_k^1 & R_5^2 + S_5^k D_k^2 & R_5^3 + S_5^k D_k^3 \\ R_6^0 + S_6^k D_k^0 & R_6^1 + S_6^k D_k^1 & R_6^2 + S_6^k D_k^2 & R_6^3 + S_6^k D_k^3 \\ R_7^0 + S_7^k D_k^0 & R_7^1 + S_7^k D_k^1 & R_7^2 + S_7^k D_k^2 & R_7^3 + S_7^k D_k^3 \end{bmatrix}$$

* Источник: https://github.com/riscv-admin/vector

В чем отличие от произведения плотных матриц?

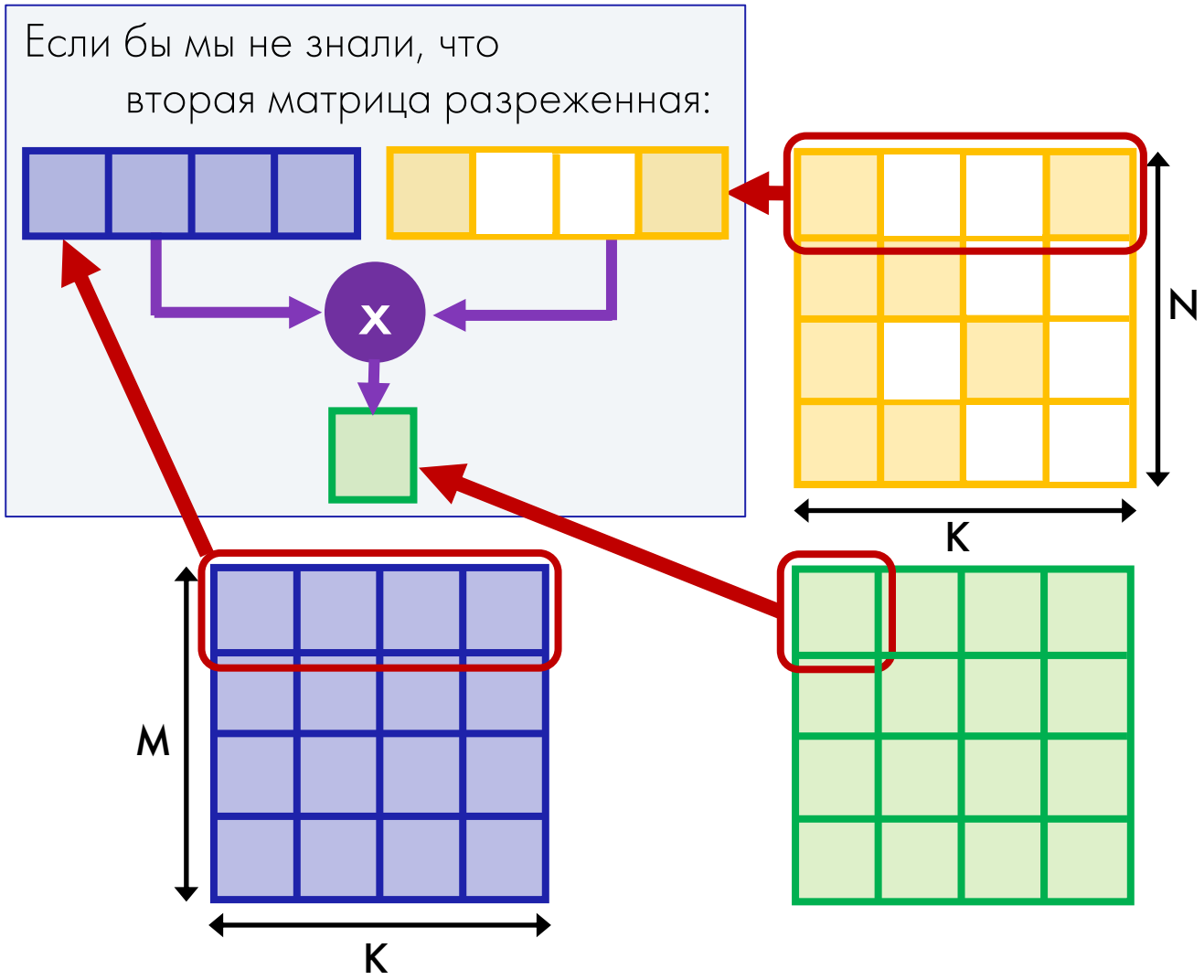
Нули в S^k не храним, не загружаем!

$$S^k \times D_k = \begin{bmatrix} S_0^k \\ S_2^k \\ S_3^k \\ S_6^k \end{bmatrix} \times \begin{bmatrix} D_k^0 & D_k^1 & D_k^2 & D_k^3 \end{bmatrix} = \begin{bmatrix} S_0^k D_k^0 & S_0^k D_k^1 & S_0^k D_k^2 & S_0^k D_k^3 \\ S_2^k D_k^0 & S_2^k D_k^1 & S_2^k D_k^2 & S_2^k D_k^3 \\ S_3^k D_k^0 & S_3^k D_k^1 & S_3^k D_k^2 & S_3^k D_k^3 \\ S_6^k D_k^0 & S_6^k D_k^1 & S_6^k D_k^2 & S_6^k D_k^3 \end{bmatrix}$$

- На блок S^k нужно меньше регистров.
- Нужно загружать маску портрета, чтобы обновлять верные строки аккумуляторов.

$$R = R + S^k \times D_k = \begin{bmatrix} R_0^0 & R_0^1 & R_0^2 & R_0^3 \\ R_1^0 & R_1^1 & R_1^2 & R_1^3 \\ R_2^0 & R_2^1 & R_2^2 & R_2^3 \\ R_3^0 & R_3^1 & R_3^2 & R_3^3 \\ R_4^0 & R_4^1 & R_4^2 & R_4^3 \\ R_5^0 & R_5^1 & R_5^2 & R_5^3 \\ R_6^0 & R_6^1 & R_6^2 & R_6^3 \\ R_7^0 & R_7^1 & R_7^2 & R_7^3 \end{bmatrix} + S^k \times D_k = \begin{bmatrix} R_0^0 + S_0^k D_k^0 & R_0^1 + S_0^k D_k^1 & R_0^2 + S_0^k D_k^2 & R_0^3 + S_0^k D_k^3 \\ R_1^0 & R_1^1 & R_1^2 & R_1^3 \\ R_2^0 + S_2^k D_k^0 & R_2^1 + S_2^k D_k^1 & R_2^2 + S_2^k D_k^2 & R_2^3 + S_2^k D_k^3 \\ R_3^0 + S_3^k D_k^0 & R_3^1 + S_3^k D_k^1 & R_3^2 + S_3^k D_k^2 & R_3^3 + S_3^k D_k^3 \\ R_4^0 & R_4^1 & R_4^2 & R_4^3 \\ R_5^0 & R_5^1 & R_5^2 & R_5^3 \\ R_6^0 + S_6^k D_k^0 & R_6^1 + S_6^k D_k^1 & R_6^2 + S_6^k D_k^2 & R_6^3 + S_6^k D_k^3 \\ R_7^0 & R_7^1 & R_7^2 & R_7^3 \end{bmatrix}$$

Разработка Sparsity Subset в XuanTie Matrix Extension



Отличия от предыдущего подхода:

1. Плотная матрица и разреженная поменялись местами:

   $R_{M\times N} = D_{M\times K} \times S_{K\times N}.$

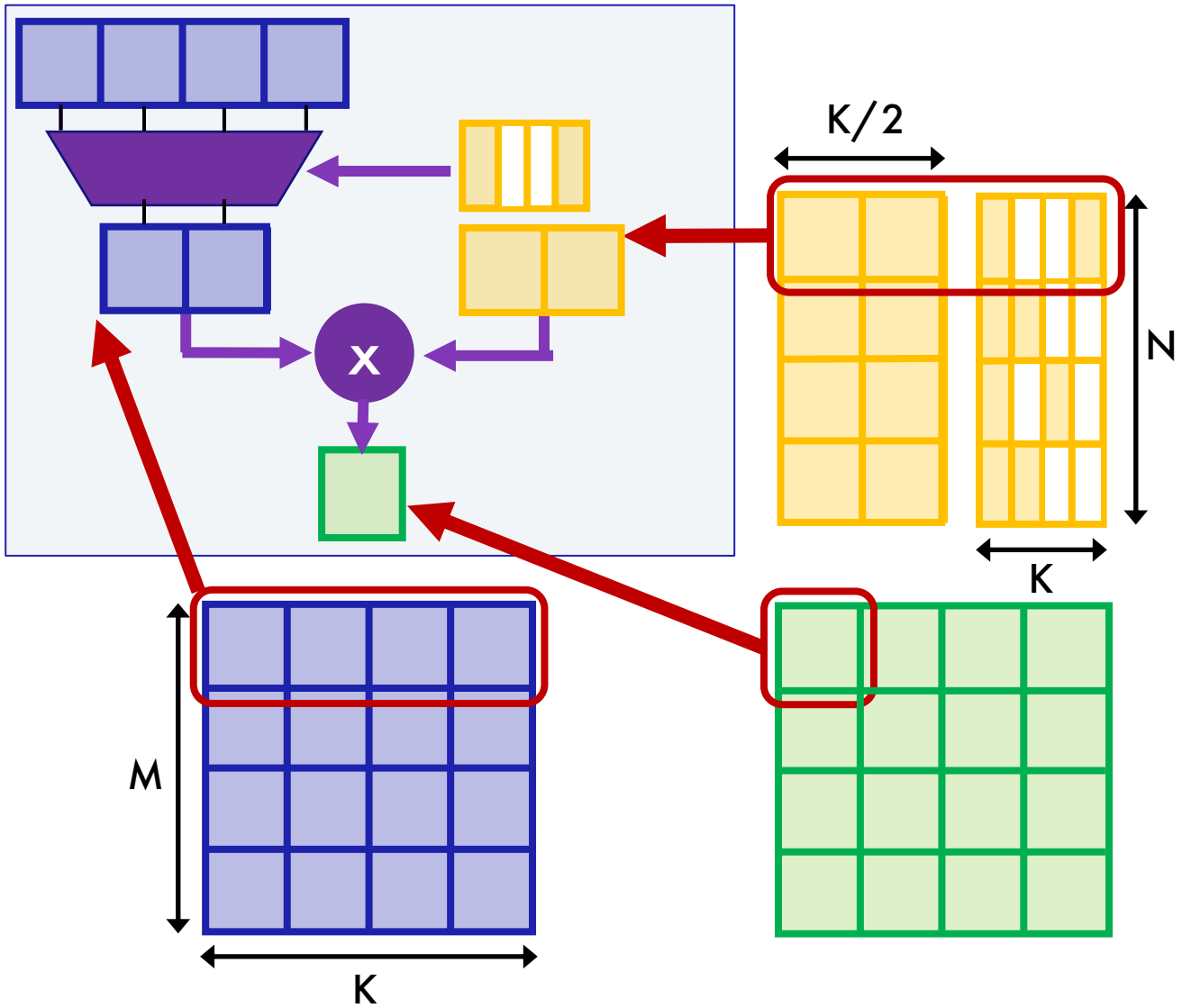
2. Разреженную матрицу предварительно транспонируем.
3. Вместо внешнего произведения векторов используем скалярное:

   $R_{M\times N} = [0]_{M\times N},$

   $\text{for k} = [0, \text{K}) \quad R_{M\times N} \mathrel{+}= D_k \cdot S^k.$

* Источник: https://github.com/riscv-admin/vector/blob/main/minutes/2023/2023-12-11/Xuantie_Matrix_Extension_about_Sparsity.pdf

Разработка Sparsity Subset в XuanTie Matrix Extension

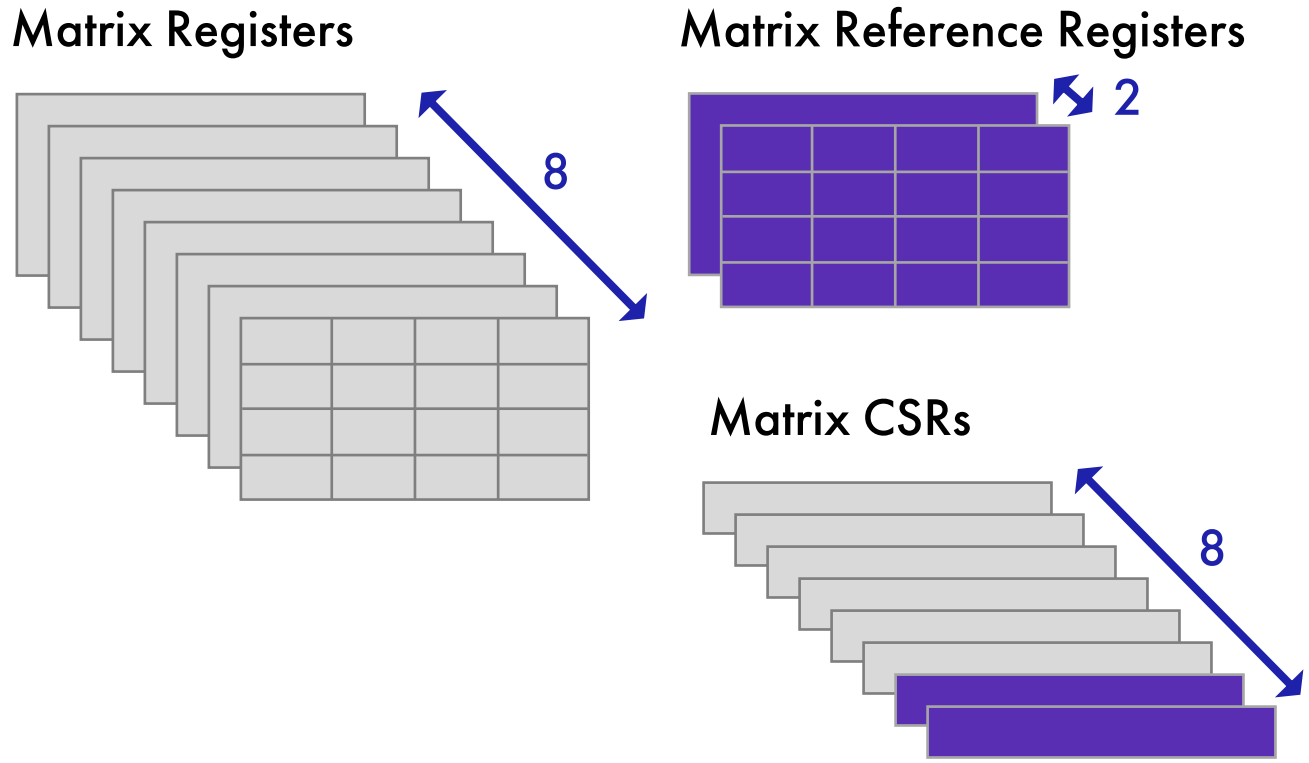


Отличия от предыдущего подхода:

1. Маска индексов портрета загружается в отдельный регистр.
2. Через маску портрета «просеиваем» не аккумуляторы, а строки плотной матрицы.
3. «Просеянную» плотную матрицу умножаем на разреженную (уже хранящуюся в формате без нулей) как обычные плотные матрицы.

* Источник: https://github.com/riscv-admin/vector/blob/main/minutes/2023/2023-12-11/Xuantie_Matrix_Extension_about_Sparsity.pdf

Дополнительные регистры и инструкции



+ **2 матричных 2D регистра** для маски индексов портрета.
+ **2 control state регистра (CSR):**
  - Число элементов в одном разреженном блоке.
  - Разреженность (доля нулевых элементов).
+ **3** инструкции **Sparse Matrix MACC.**
+ **6** инструкций **Memory access** для регистров маски индексов портрета.

* Источник: https://github.com/riscv-admin/vector/blob/main/minutes/2023/2023-12-11/Xuantie_Matrix_Extension_about_Sparsity.pdf

Определения и обстоятельства

Разработка T-Head RVM ⟶ RISC-V AME TG

Предложения RISC-V Vector SIG ⟶ RISC-V IME TG

Свежие вести с полей: Sparse Lives Matter

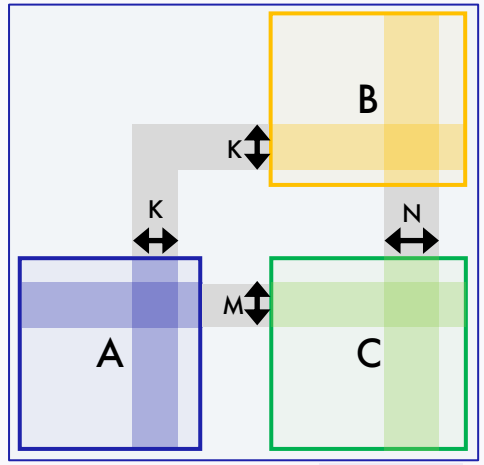
Заключение

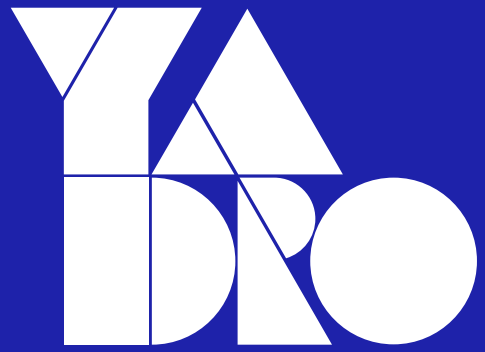
Заключение

- Матричные расширения стали появляться не так давно – **с 2020 года.**
  В этом плане x86, Power, Arm и RISC-V почти наравне.
- Подходят для ускорения AI/ML, AR/VR, CV, ADAS, HPC.
- Хороший ход – поддержка в **OpenBLAS** и **Eigen** (как у Power MMA).
- Стандартные матричные расширения RISC-V: ← **~ в середине 2025 года**
  - **RISC-V Attached Matrix Extension (T-Head via AME TG):**
    - specification https://github.com/T-head-Semi/riscv-matrix-extension-spec
    - 20+ matrix instructions, 8 matrix registers
    - AI/ML Data Types
    - ~9.5x performance (demos)
  - **RISC-V Integrated Matrix Extension (Vector SIG via IME TG):**
    - HPC GEMM (BLIS microkernel)
    - HPC & AI/ML Data Types
    - Goal: ~90% peak performance (GEMM); ~2x performance (vs vector).
  - **Поддержке разреженных матриц быть!**

(!) Обновления на Github:

- Vector SIG
- IME TG
- AME TG





Москва,
ул. Рочдельская, 15, стр. 13
+7 800 777-06-11

yadro.com