Цикловые оптимизации и векторизация в RISC-V

November 6, 2024

Konstantin Vladimirov, Mark Goncharov
info@syntacore.com

RISC-V International

RISC-V это свободная и открытая RISC ISA

- Является результатом общемирового сотрудничества

RISC-V Foundation / International

- Основан в **2015** индустриальными лидерами и стартапами
- **3900 +** членов из **70 +** стран
- Продвигает исследования и инновации

Российский Альянс RISC-V

- Создан для развития и популяризации архитектуры **RISC-V** в нашей стране, представления и защиты общих интересов участников
- **Основная цель** – создание открытого сообщества разработчиков программного и аппаратного обеспечения

Преимущества RISC-V

Простота

- Базовый набор команд существенно меньше, чем другие коммерческие ISA

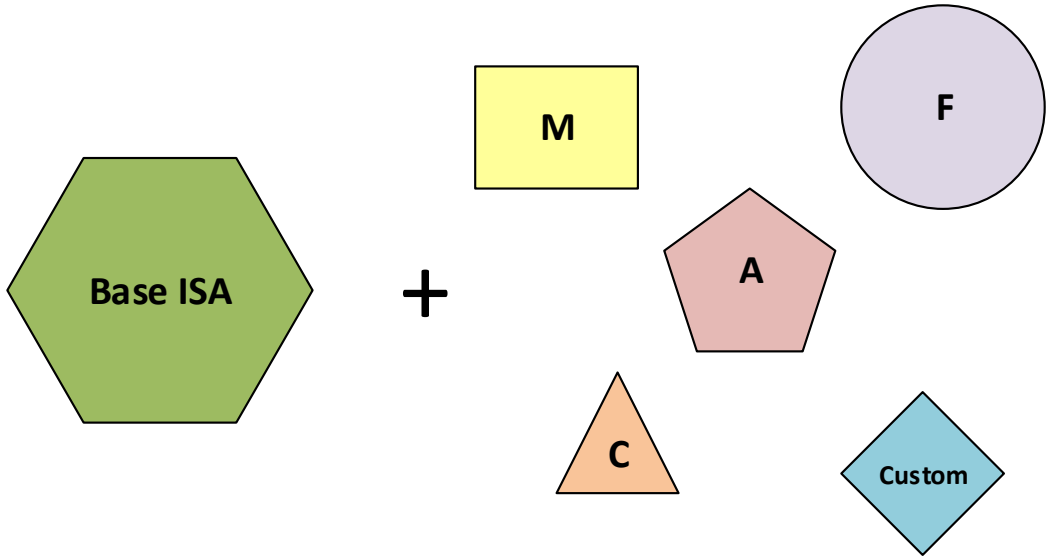
Модульная структура с поддержкой расширяемости и специализации

- Широкий набор стандартных расширений
- Разумное управление кодированием команд, существенное резервирование

Стабильность

- Базовый набор и стандартные расширения зафиксированы
- Добавление функциональности через расширения, не выпуск новых версий

Спецификации доступны для свободного и бесплатного использования



Базовый целочисленный набор команд (на выбор)

- RV32I – для работы с 32-битными целыми и адресами
- RV32E – вариация RV32I с урезанным количеством регистров общего назначения
- RV64I, RV128I – наборы команд для поддержки расширенных целых и 64-битных адресов

Case study: векторизуем find?

```
int find(const int *a, int n, int x) {
  for (int i = 0; i < n; i++)
    if (a[i] == x)
      return i;
  return -1;
}
```

- Давайте прежде чем рассматривать подход RISC-V рассмотрим более распространённый вариант векторизации с фиксированным размером SIMD, как в AVX.
- Есть ли у нас идеи что мы тут можем выиграть фиксированной векторизацией?

Ручная векторизация find на AVX

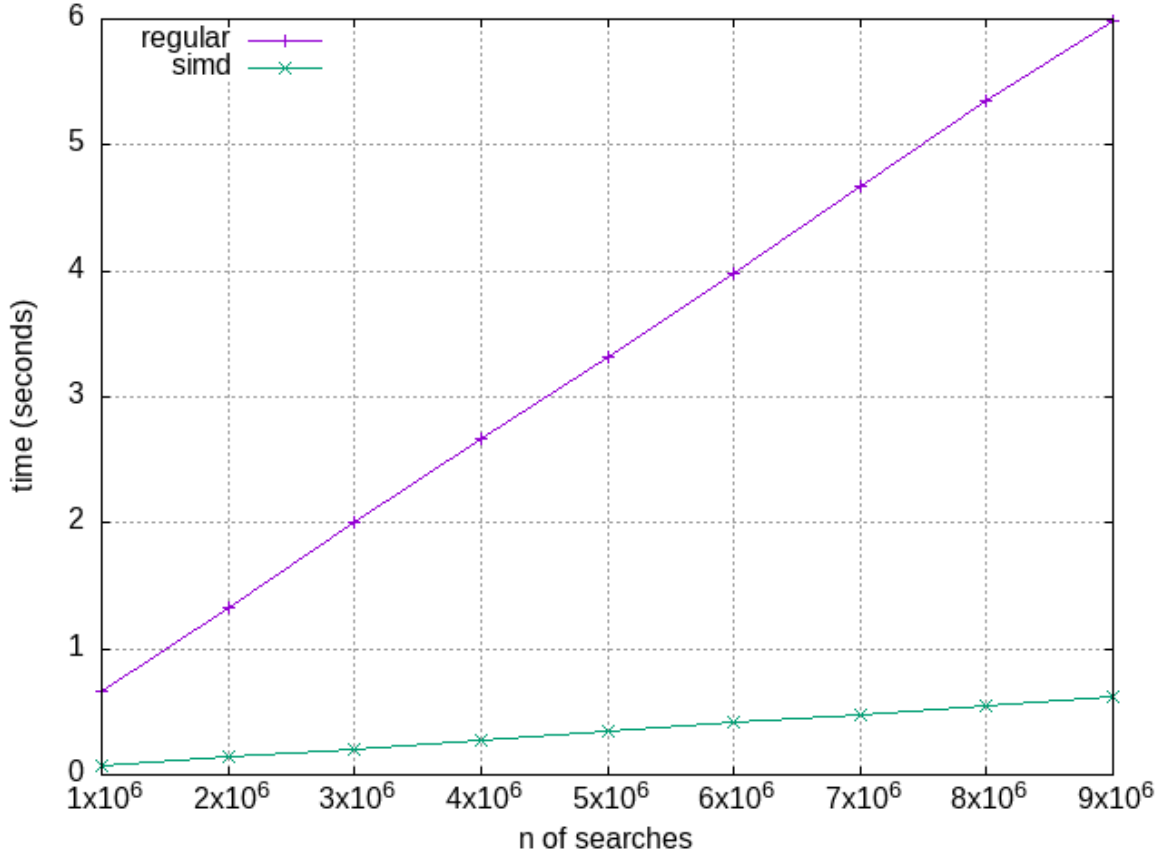
```
int find_simd(const int *a, int n, int x) {
  if (n >= 16 && __builtin_cpu_supports("avx512f")) {
    __m512i needle = _mm512_set1_epi32(x);
    for (int i = 0; i < (n / 16) * 16; i += 16) {
      __m512i current = _mm512_loadu_si512(a + i);
      __mmask16 m =
          _mm512_cmp_epi32_mask(needle, current, _MM_CMPINT_EQ);
      if (m != 0)
        return i + __builtin_ctz(m);
    }
  }
  // дальше обработка хвоста
```

Find если писать его руками

```
int find_simd(const int *a, int n, int x) {
  if (n >= 16 && /* cpu supports zmm registers */) {
    // 64-byte SIMD implementation
  }
  if (n >= 8 && /* cpu supports ymm registers */) {
    // 32-byte SIMD implementation
  }
  if (n >= 4 && /* cpu supports xmm registers */) {
    // 16-byte SIMD implementation
  }
  while (n >= 1) {
    // 4-byte tail (non-vectorized find)
  }
}
```

Стоит ли овчинка выделки?

- С одной стороны мы идём на чудовищное усложнение кода.
- Часто его даже пишут на ассемблере в максимально ужасном стиле (посмотрите memcpy в glibc).
- С другой стороны выгода очевидна и люди на это идут.
- Более того – на это идут разработчики hardware, постоянно усложняя его новыми векторными возможностями.



Справится ли компилятор для AVX?

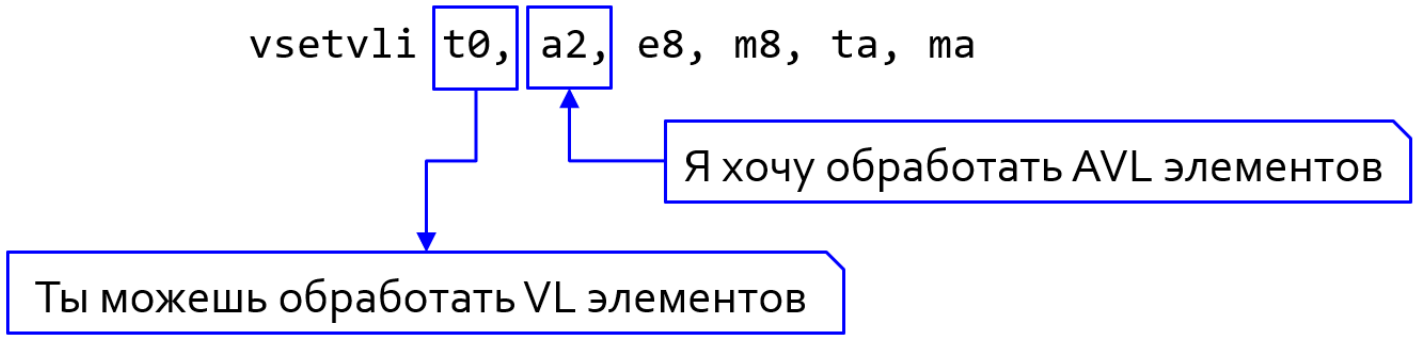
- Ни gcc 14 ни clang 19 [не векторизуют](#) find по умолчанию даже при максимальных оптимизациях.
- Они кое-как справляются с memcpy, порождённый асемблер приведён справа.

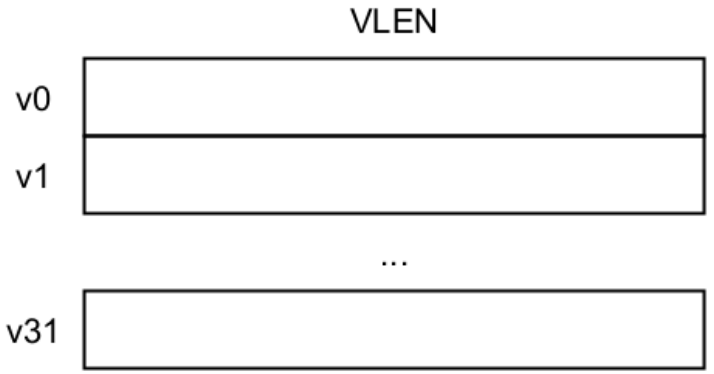
```
vmovups ymm0, [rsi + rdx]
....
vmovups ymm3, [rsi + rdx + 96]
vmovups [rdi + rdx], ymm0
....
vmovups [rdi + rdx + 96], ymm3
```

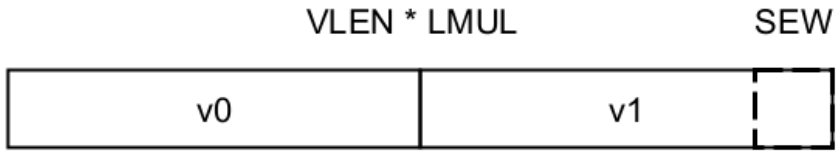
```
memcpy(void*, void*, int):                # @m
        test    edx, edx
        jle     .LBB0_16
        mov     eax, edx
        xor     ecx, ecx
        cmp     edx, 16
        jb      .LBB0_12
        mov     r8, rdi
        sub     r8, rsi
        cmp     r8, 128
        jb      .LBB0_12
        cmp     edx, 128
        jae     .LBB0_5
        xor     ecx, ecx
        jmp     .LBB0_9
.LBB0_5:
        mov     ecx, eax
        and     ecx, -128
        xor     edx, edx
.LBB0_6:                                  # =>This Inner
        vmovups ymm0, ymmword ptr [rsi + rdx]
        vmovups ymm1, ymmword ptr [rsi + rdx + 32]
        vmovups ymm2, ymmword ptr [rsi + rdx + 64]
        vmovups ymm3, ymmword ptr [rsi + rdx + 96]
        vmovups ymmword ptr [rdi + rdx], ymm0
        vmovups ymmword ptr [rdi + rdx + 32], ymm1
        vmovups ymmword ptr [rdi + rdx + 64], ymm2
        vmovups ymmword ptr [rdi + rdx + 96], ymm3
        sub     rdx, -128
        cmp     rcx, rdx
        jne     .LBB0_6
        cmp     rcx, rax
        je      .LBB0_16
        test    al, 112
        je      .LBB0_12
.LBB0_9:
        mov     rdx, rcx
        mov     ecx, eax
        and     ecx, -16
.LBB0_10:                                 # =>This Inner
        vmovups xmm0, xmmword ptr [rsi + rdx]
        vmovups xmmword ptr [rdi + rdx], xmm0
        add     rdx, 16
        cmp     rcx, rdx
        jne     .LBB0_10
        cmp     rcx, rax
        je      .LBB0_16
.LBB0_12:
        mov     rdx, rcx
        not     rdx
        add     rdx, rax
        mov     r8, rax
        and     r8, 7
        je      .LBB0_14
.LBB0_13:                                 # =>This Inner Loop Header: Depth=1
        movzx   r9d, byte ptr [rsi + rcx]
        mov     byte ptr [rdi + rcx], r9b
        inc     rcx
        dec     r8
        jne     .LBB0_13
.LBB0_14:
        cmp     rdx, 7
        jb      .LBB0_16
.LBB0_15:                                 # =>This Inner Loop Header: Depth=1
        movzx   edx, byte ptr [rsi + rcx]
        mov     byte ptr [rdi + rcx], dl
        movzx   edx, byte ptr [rsi + rcx + 1]
        mov     byte ptr [rdi + rcx + 1], dl
        movzx   edx, byte ptr [rsi + rcx + 2]
        mov     byte ptr [rdi + rcx + 2], dl
        movzx   edx, byte ptr [rsi + rcx + 3]
        mov     byte ptr [rdi + rcx + 3], dl
        movzx   edx, byte ptr [rsi + rcx + 4]
        mov     byte ptr [rdi + rcx + 4], dl
        movzx   edx, byte ptr [rsi + rcx + 5]
        mov     byte ptr [rdi + rcx + 5], dl
        movzx   edx, byte ptr [rsi + rcx + 6]
        mov     byte ptr [rdi + rcx + 6], dl
        movzx   edx, byte ptr [rsi + rcx + 7]
        mov     byte ptr [rdi + rcx + 7], dl
        add     rcx, 8
        cmp     rax, rcx
        jne     .LBB0_15
.LBB0_16:
        vzeroupper
        ret
```

Основная идея для RVV

```
vsetvli rd, rs1, vtype1    # AVL = x[rs1], x[rd] = vl
vsetivli rd, uimm, vtype1  # AVL = uimm, x[rd] = vl
vsetvl rd, rs1, rs2        # AVL = x[rs1], VT = x[rs2], x[rd] = vl

vset* rd, x0, ...          # AVL = VLMAX
```







Повторим метод find

```
int find_rvv(const int *src, int n, int x) {
  size_t vl = vsetvl_e32m8(n);
  vint32m8_t n = vmv_v_x_i32m8(x, vl);

  for (int i = 0; n > 0; n -= vl, src += vl, i += vl) {
    vl = vsetvl_e32m8(n);
    vint32m8_t x = vle32_v_i32m8(src, vl);
    vbool4_t result = vmsne_vv_i32m8_b4(x, n, vl);
    long bitnum = vfirst_m_b4(result, vl);
    if (bitnum != -1)
      return i + bitnum;
  }

  return -1;
}
```

Обсуждение

- С такого рода векторизацией компилятор гораздо легче справляется (потенциально).
- Как мы можем ему помочь?

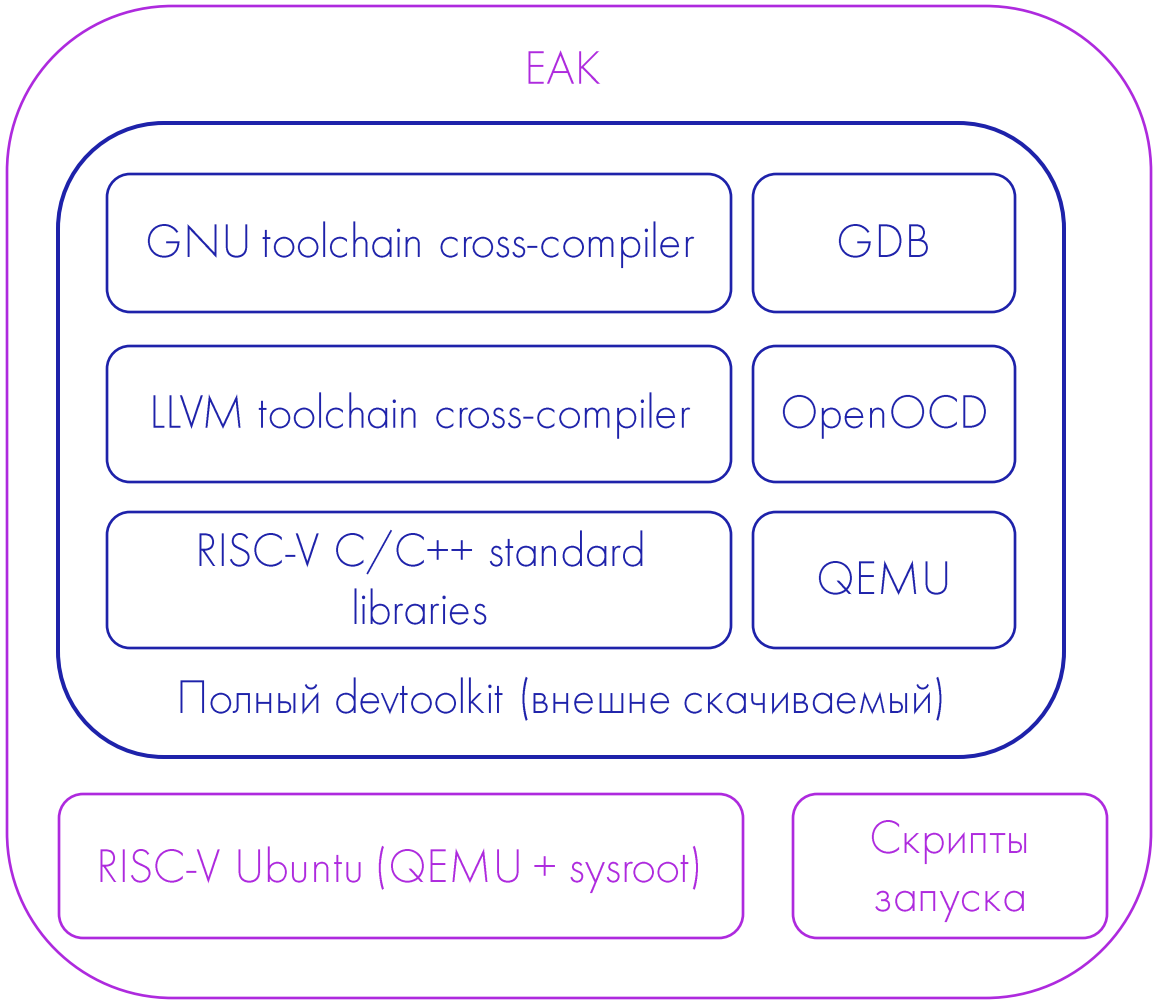
Devtoolkit

- Внешние релизы выходят раз в квартал.
- LLVM-based toolchain содержащий все SCR-специфичные опции и доработки.
- GCC-based toolchain распространяется не модифицированным.
- В релиз входят также отладчики: GDB, OpenOCD, планируется LLDB.
- Базовое системное ПО: baremetal HAL, загрузчики, Linux kernel с поддержкой SCRx.
- Функциональный симулятор QEMU для процессорного кластера.
- Примеры приложений, IDE и многое другое.

https://syntacore.com/page/products/sw-tools

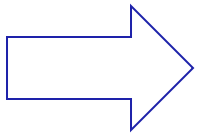
Early access kit

- Пакет обновляется чаще, чем официальные релизы.
- Полный набор инструментов и удобство использования.
- Распространяется без гарантий и поддержки.
- Идеальное решение для образования и экспериментов.
- Получить доступ можно, направив запрос по электронной почте на адрес edu@riscv-alliance.ru



Сплиттинг циклов

```
void before(int bound, int N) {
  for (int i = 0; i < N; ++i)
    if (i < bound)
      foo(i);
    else
      bar(i);
}
```



```
void after(int bound, int N) {
  int tmp = min(bound, N);
  for (int i = 0; i < tmp; ++i)
    foo(i);
  for (int i = tmp; i < N; ++i)
    bar(i);
}
```

- Позволяет векторизации более эффективно обрабатывать циклы.
- Дополнительные оптимизации вроде loop distribution могут сделать результат ещё несколько лучше.

Реалистичный пример из hmmer: loop splitting

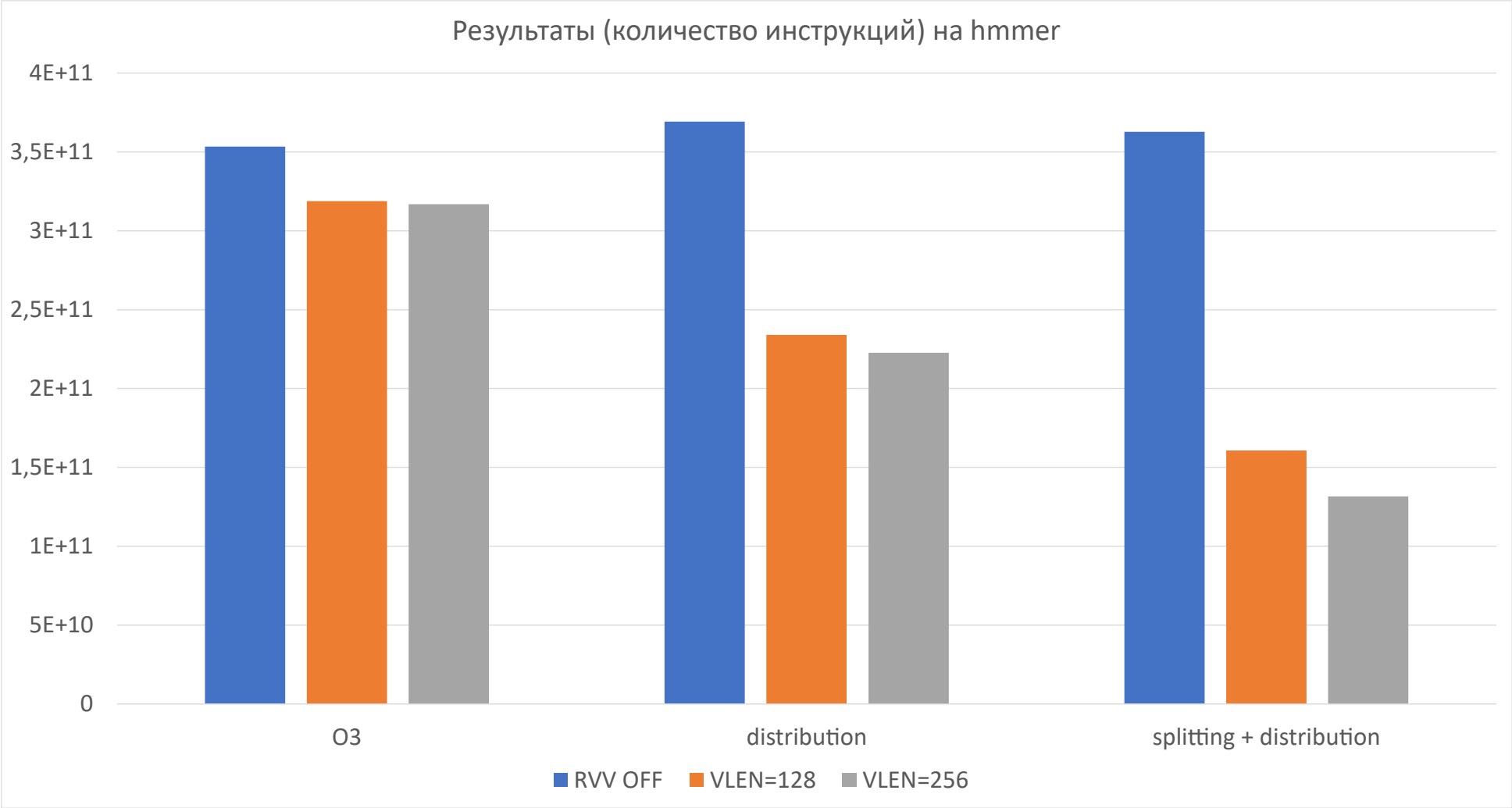
```
for (k = 1; k <= M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
  if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
    mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])
    mc[k] = sc;
  if ((sc = xmb  + bp[k]) > mc[k])
    mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;

  if (k < M) {
    ic[k] = mpp[k] + tpmi[k];
    if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
    ic[k] += is[k];
    if (ic[k] < -INFTY) ic[k] = -INFTY;
  }
}
```

```
for (k = 1; k < M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
  if ((sc = ip[k-1] + tpim[k-1]) > mc[k])
    mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])
    mc[k] = sc;
  if ((sc = xmb  + bp[k]) > mc[k])
    mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;

  ic[k] = mpp[k] + tpmi[k];
  if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
  ic[k] += is[k];
  if (ic[k] < -INFTY) ic[k] = -INFTY;
}

// далее хвост после loop splitting
```

Реалистичный пример из hmmer: loop distribution

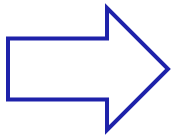
```
for (k = 1; k < M; k++) {
  mc[k] = mpp[k-1] + tpmm[k-1];
  if ((sc = ip[k-1] + tpim[k-1]) > mc[k])    mc[k] = sc;
  if ((sc = dpp[k-1] + tpdm[k-1]) > mc[k])    mc[k] = sc;
  if ((sc = xmb  + bp[k]) > mc[k])    mc[k] = sc;
  mc[k] += ms[k];
  if (mc[k] < -INFTY) mc[k] = -INFTY;

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k]) dc[k] = sc;
  if (dc[k] < -INFTY) dc[k] = -INFTY;

  ic[k] = mpp[k] + tpmi[k];
  if ((sc = ip[k] + tpii[k]) > ic[k]) ic[k] = sc;
  ic[k] += is[k];
  if (ic[k] < -INFTY) ic[k] = -INFTY;
}

// далее хвост после loop splitting
```

Результаты для обоих пассов



Векторизация математических функций

```
void before(int bound, int N) {
  for (int i = 0; i < size; ++i)
    b[i] = log(a[i]);
}
```

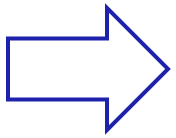


```
void after(int bound, int N) {
  for (int i = 0;
       i < size; i += vl)
    vectorized_log(a, b, i);
}
```

- На самом деле компилятор не построит тут вызов, а подставит векторизованный логарифм.

Векторизация математических функций RISC-V

```
void before(int bound, int N) {
  for (int i = 0; i < size; ++i)
    b[i] = log(a[i]);
}
```

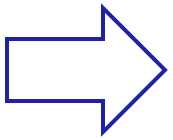


```
double a[n], b[n];
for (int i = 0; i < n; i += vl) {
  vl = getVL(n - i);
  vdouble va = vload(a + i, vl);
  vdouble vb = vlog(va);
  vstore(b + i, vb, vl);
}
```

- На самом деле компилятор не построит тут вызов, а подставит векторизованный логарифм.
- Это платформенно-специфичное решение и для разных архитектур компилятор подставит разные реализации

Подстановка векторных интринсиков

```
double a[n], b[n];
for (int i = 0; i < n; i += vl) {
  vl = getVL(n – i);
  vdouble va = vload(a + i, vl);
  vdouble vb = vlog(va);
  vstore(b + i, vb, vl);
}
```



```
for (int i = 0; i < n; i += vl) {
  vl = vsetvl_e64m1(n - i);
  vfloat64m1_t va =
    vle64_v_f64m1(a + i, vl);
  vfloat64m1_t vb = vlog(va);
  vse64_v_f64m1(b + i, vb, vl);
}
```

- К сожалению не всё можно заменить на интринсики, в RISC-V просто нет интринсика для vlog.
- Такой подход требует, чтобы функция vlog уже существовала в какой-то библиотеке и эта библиотека была доступна при линковке.
- Разные компании решают это по разному, например известны библиотеки Intel SVML, ARM PL, GNU-библиотека libmvec и прочие.

Введение в библиотеку SLEEF

- Векторные реализации
  - Libm
  - Libm quad precision
  - Discrete Fourier transform
- Поддержка архитектур
  - RISC-V : RVV
  - x86 : SSE, AVX
  - AArch : NEON, SVE
  - PowerPC : VSX
  - System/390 : VXE
  - WebAssembly : SSE2
  - CUDA

porting-sleef-to-risc-v-ludovic-henry-rivos-eric-love-sifive

Введение в библиотеку SLEEF

- Векторные реализации
  - Libm
  - Libm quad precision
  - Discrete Fourier transform
- Поддержка архитектур
  - RISC-V : RVV
  - x86 : SSE, AVX
  - AArch : NEON, SVE
  - PowerPC : VSX
  - System/390 : VXE
  - WebAssembly : SSE2
  - CUDA

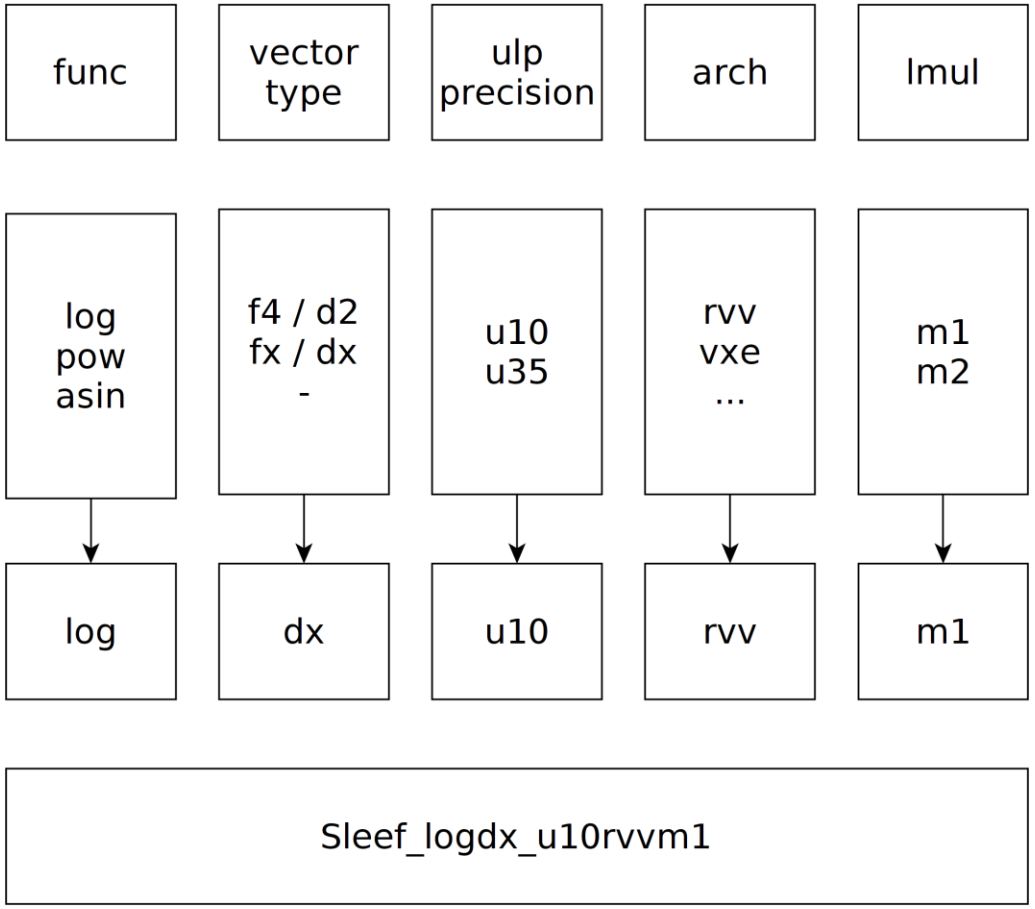
```
VECTOR_CC vdouble xlog(vdouble d) {
 // ...
 if (vlt_vd_vd(vabs_vd(d), s))
   vdouble dql = vrint_vd_vd(
     vmul_vd_vd_vd(d, vcast_vd_d(M_1_PI)));
 // ...
```

Замены для RISC-V (для случая LMUL = 1)

```
vdouble        ⇒ vfloat64m1_t
vlt_vo_vd_vd   ⇒ __riscv_vmflt
vrint_vd_vd    ⇒ __riscv_vfcvt_f_x_v_f64m1
vmul_vd_vd_vd  ⇒ __riscv_vfmul
vcast_vd_d     ⇒ __riscv_vfmv_v_f_f64m1
```

Разнообразие функций SLEEF

- Поддержаны различные функции
- Для каждой из них может быть разный векторный тип
- Для каждой из них может быть разная точность
- Они поддержаны под разные архитектуры
- Также учитывается специфика, например для RVV это LMUL, SEW, etc
- В итоге один алгоритм раскрывается в несколько десятков функций.



Возможности для оптимизации

- Рассмотрим обобщённую функцию внутри логарифма.

```
vmul_vd_vd_vd(d, vcast_vd_d(M_1_PI));
```

- Простая замена каждой обобщённой функции на интринсик может порождать лишние ассемблерные инструкции.

```
vcast_vd_d     ⇒  __riscv_vfmv_v_f_f64m1  ⇒  vfmv.v.f v10, tab
vmul_vd_vd_vd  ⇒  __riscv_vfmul_vv_f64m1  ⇒  vfmul.vv v8, v8, v10
```

- Например, в случае RISC-V мы хотели бы сопоставить этому коду вызов всего одной векторной инструкции.

```
vfmul.vf   v8, v8, tab
```

- Для этого можно улучшать компиляторные оптимизации на виртуальных регистрах.

Требование к компилятору

- Исходный код.

```
for (int i = 0; i < size; ++i)
    b[i] = log(a[i]);
```

- Трансформированный вариант с учётом вызова SLEEF.

```
for (int i = 0; i < n; i += vl) {
  vl = __riscv_vsetvl_e64m1(n - i);
  vfloat64m1_t va = __riscv_vle64_v_f64m1(a + i, vl);
  vfloat64m1_t vb = Sleef_logdx_u10rvvm1(va);
  __riscv_vse64_v_f64m1(b + i, vb, vl);
}
```

- Основная проблема для компилятора – доказать корректность и оценить эффект.

Поддержка в SC-Devtoolkit и upstream LLVM

- В Syntacore devtoolkit Nov'24 будет добавлена поддержка SLEEF для RISC-V.
- Мы также сейчас находимся в процессе апстриминга этой поддержки.

```
for (int i = 0; i < size; ++i)
  a[i] = log(a[i]);
```

```
clang
  -O2
  -march=rv64gcv
  -fveclib=SLEEF
  -fno-math-errno
```

```
.preheader:
   csrr        a2, vlenb
.vector.body:
   # v8, v9 = a[i, i+1, ...]
   vl2re64.v   v8, (s1)
   call        Sleef_logdx_u10rvvm2
   sub         s0, s0, s5 # i = i - vl * 2
   vs2r.v      v8, (s1)
   add         s1, s1, s6 # a = a + vl * 2
   bnez        s0, .vector.body
```

https://github.com/llvm/llvm-project/pull/114014

LSR Termination Folding

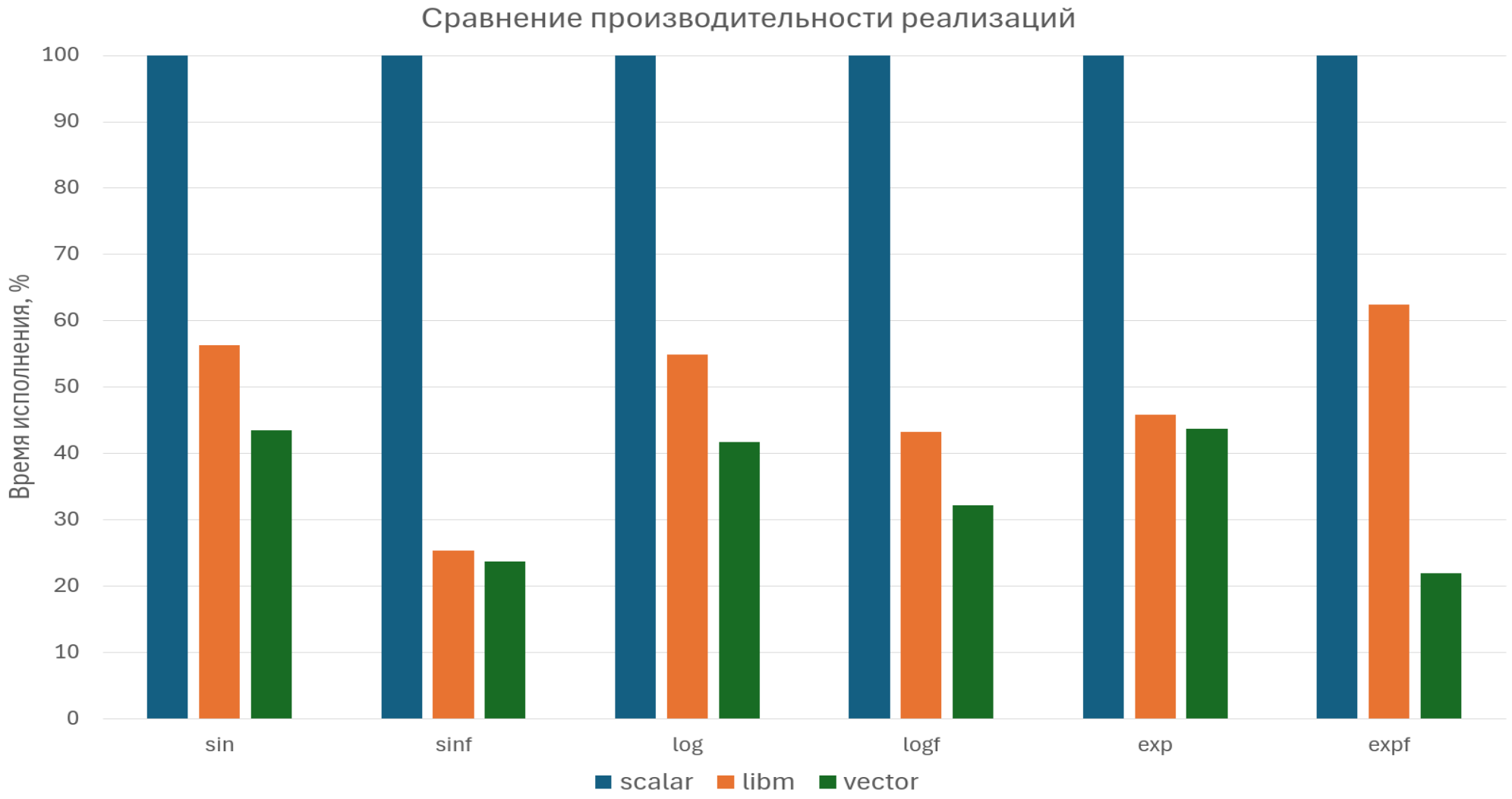
- Оптимизация, позволяющая сворачивать индуктивные переменные, изменяя условие выхода из цикла.
- Использует информацию из Scalar Evolution анализа (SCEV).

```
for (int i = 0; i < size; ++i)
  a[i] = log(a[i]);
```

```
clang
  -O2
  -march=rv64gcv
  -fveclib=SLEEF
  -fno-math-errno
  -mllvm -lsr-term-fold
```

```
.preheader:
   csrr        a2, vlenb
.vector.body:
   # v8, v9 = a[i, i+1, ...]
   vl2re64.v   v8, (s1)
   call        Sleef_logdx_u10rvvm2
   vs2r.v      v8, (s1)
   add         s1, s1, s6
   blt         s1, s0 .vector.body
```

Результаты



Общий вывод

- Векторные расширения позволяют компиляторам более агрессивно оптимизировать код за счёт отказа от обработки хвоста.
- Аппаратное обеспечение с поддержкой RVV широко появляется у разных вендоров.
- Syntacore Devtoolkit поддерживает cutting edge оптимизации, часть из которых попадает в upstream LLVM.
- Библиотека SLEEF уже сейчас позволяет эффективно векторизовать математические вычисления.
- Будущее остаётся за открытыми технологиями и открытыми экосистемами.

Syntacore™

Custom cores and tools

ВСЕМ СПАСИБО, А ТЕПЕРЬ Q&A