Soft Machines™

Оптимизирующий Компилятор

на базе UTL и LLVM для ARM и MIPS

Александр Юльевич Дроздов
Виктор Евгеньевич Владиславлев
Святослав Владимирович Кузьмич

Открытая Конференция по Компиляторным Технологиям

Декабрь 2015, Москва

Содержание

- Предпосылки
- Универсальная Библиотека Трансляции (UTL)
- Трансформации
  - Оптимизации доступа в память
  - Авто-Распараллеливание
  - Векторизация
- Производительность
  - Smart Compiler (SMCC)
  - Замеры на ARM
  - Замеры на MIPS

Предпосылки

ARM для серверов

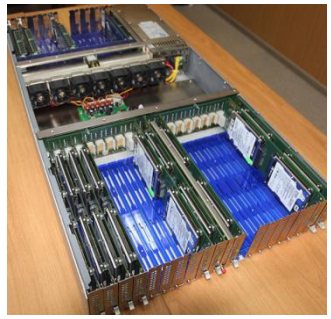
- **Qualcomm** анонсировала **24-ядерный ARM** для серверов
- **Cavium**, **Broadcom**, **Applied Micro**, **AMD** и проч. сделали это ранее
- **Baidu**: x86 -> ARM уменьшила общие издержки на **25%**
- **Rikor** создала системы хранения данных на основе ARM серверов
- **Linaro**: сообщество с девизом «open source software for ARM SoCs»

Dell

HP

Возрождение MIPS

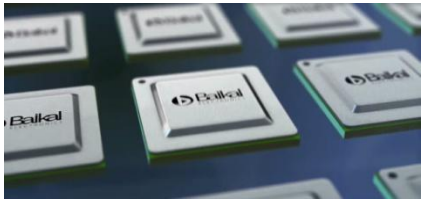
- **Imagination**™ заявлял: MIPS займет до 30% от рынка ARM
- Ядра **Warrior**– “new architecture and best-in-class performance”
- **Baikal Electronics** – первый разразботчик процессоров на основе Warrior P5600



Made in Russia

Development Tools, Compiler firstly

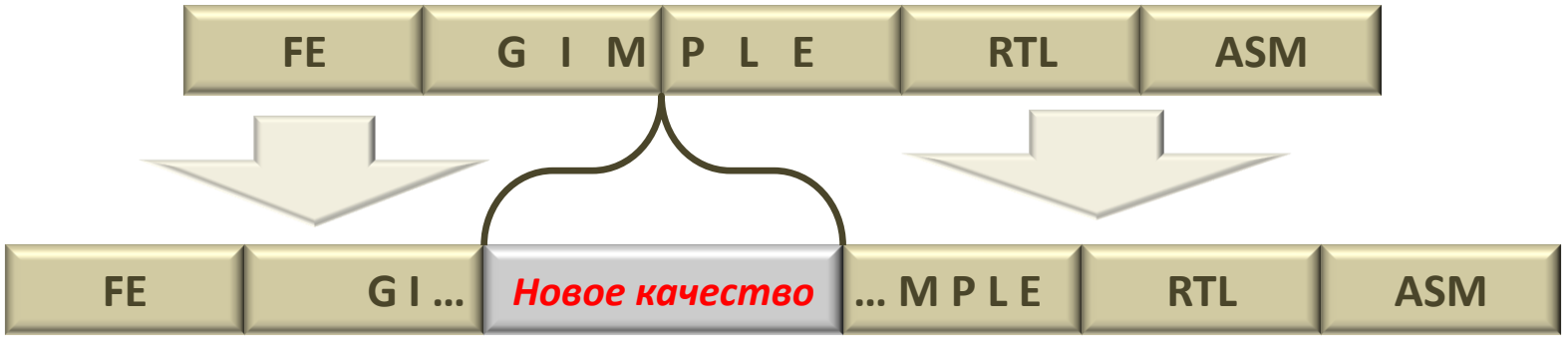
- Для x86 есть выдающиеся компиляторы от Intel or Sun
- Для ARM сообщество докручивает GNU решение – linaro-gcc
- MIPS – довольствуется базовым портом GCC и LLVM



Baikal-T1

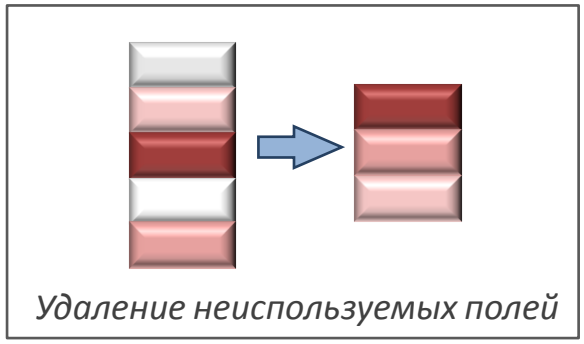
Универсальная Библиотека Трансляции

- Большой опыт в области оптимизирующей компиляции (Sun, Elbrus, Intel, GNU, LLVM) воплотился в *Компонентная Технология Оптимизирующей Трансляции* (**КТОТ**) – подход, который абстрагирует алгоритмы анализа и оптимизаций от конкретного промежуточного представления
- УБТ (или UTL) – реализация КТОТ – была апробирована:
  - в составе GNU и LLVM цепочек трансляции
  - для архитектур x86, ARM, MIPS, Itanium, Power, и Cell B.E
  - в двоичном трансляторе QEMU – x86->ARM
  - в проприетарном JIT-компиляторе SMI
- UTL
  - Может применяться в разных пользовательских цепочках трансляции
  - Имеет весь основной инструментарий компиляторостроения
  - Содержит Алгоритмы анализа и Оптимизаций
  - На его основе реализованы Авто-распараллеливатель и Векторизатор

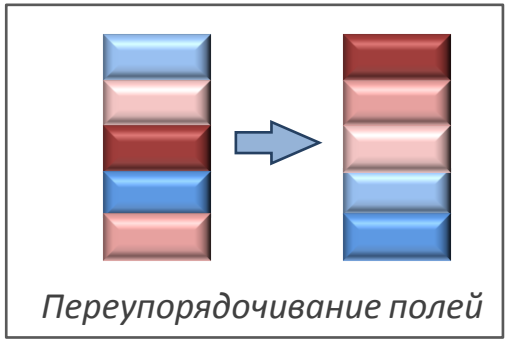


Глобальные Трансформации Данных `struct`

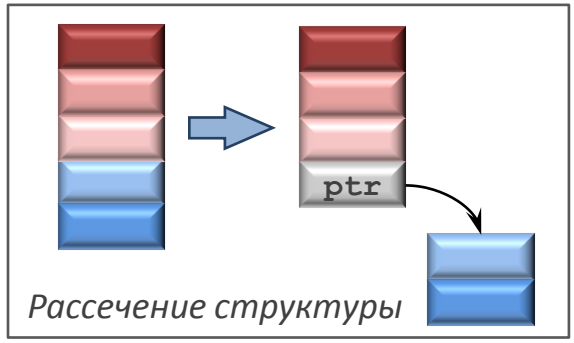
- Направлены на повышение *локальности* данных
- Требуют режима «*вся программа*» (whole program mode)
- Необходим сложный анализ – Data Structure Analysis (в LLVM до 2.8)
- На **429.mcf**, **462.libquantum** и **179.art** дает 50-70%



Удаление неиспользуемых полей

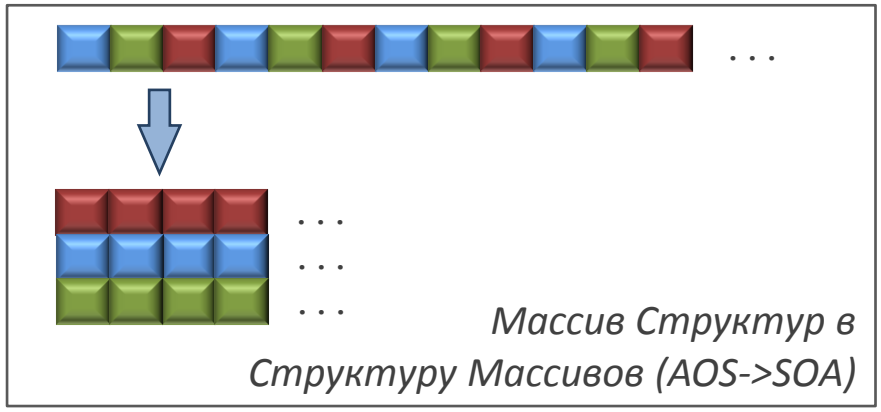


Переупорядочивание полей



Рассечение структуры

429.mcf

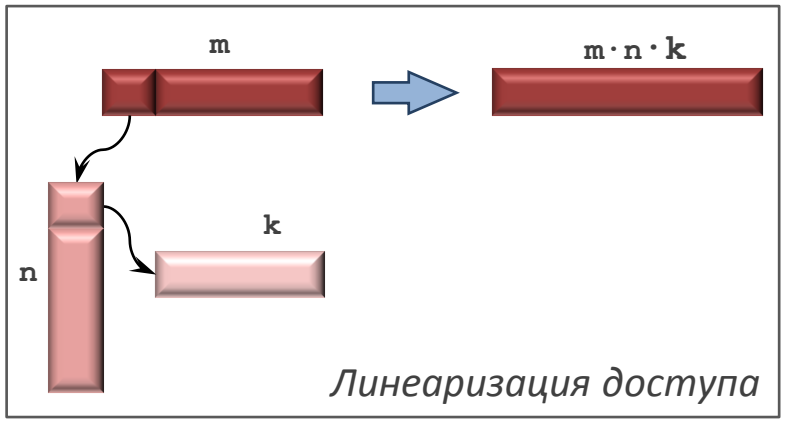


Массив Структур в Структуру Массивов (AOS->SOA)

462.libquantum
179.art

Глобальные Трансформации Данных array

- Линеаризация доступа – убираем промежуточные LOADs
- Необходим контроль нового – линеаризованного – индекса на переполнение
- Требуется whole program mode
- Классика: на **183.equake** дает 100%



Линеаризация доступа

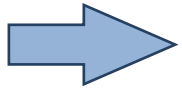
- Анализ многоуровневой аллокации полезен в **464.h264ref**
- Доказывает независимость STOREs, что дает возможность распараллелить цикл

```
statis int *****BlockSAD;
BlockSAD = malloc();
BlockSAD[i] = malloc();
BlockSAD[i][j] = malloc();
BlockSAD[i][j][k] = malloc();
BlockSAD[i][j][k][l] = malloc();...
for(pos=0;pos<max_pos;pos++) { ...
  for(blky=0;blky<4;blky++) { ...
    BlockSAD[list][ref][7][ind++][pos]+= …
    BlockSAD[list][ref][7][ind++][pos]+= …
  }
}
```

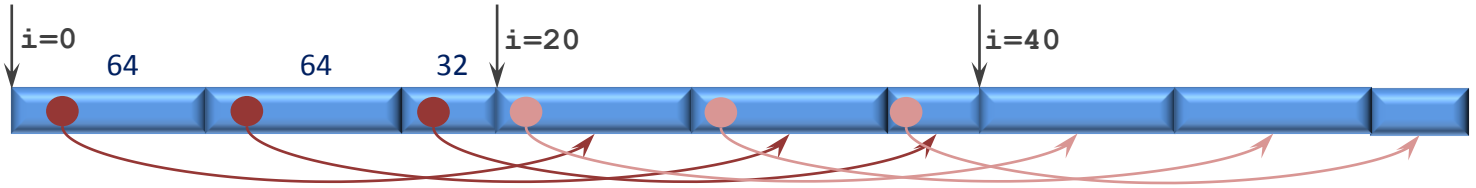
Предподкачка

- Направлена на уменьшение промахов в кэш
- Требует аппаратной поддержки – спец. инструкций
- Результат сильно зависит от архитектуры и ее подсистемы памяти
- Не требует глобального контекста
- На **470.lbm**:
  - **Cortex-A15** Chromebook – **30%**
  - **Aarch64** – **18%**
  - **MIPS** – несколько процентов

```
for(;;i+=20) {
 = srcGrid[i];...
 = srcGrid[i+19];
...
}
// sizeof(srcGrd[0]) is 8
```



```
for(;;i+=20) {
 prefetch( srcGrid[i+20]);
 prefetch( srcGrid[i+28]);
 prefetch( srcGrid[i+23]);
= srcGrid[i];... = srcGrid[i+19];
}
```



Cacheline size – 64 bytes

Авто-распараллеливание

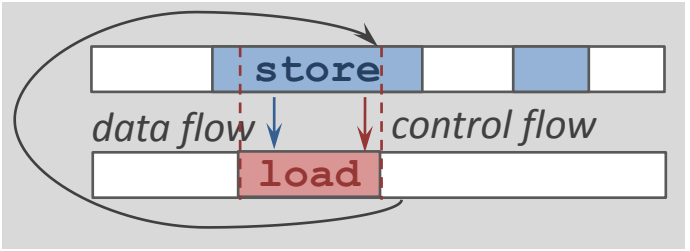
- Параллелизм – секционный и **циклический**
- Чтобы распараллелить цикл достаточно отсутствие *кросс-итерационных* зависимостей
- Редукции – возможны; но надо помнить о **fast-math** (**183**, **435**, **465**)
- Swap-рекуррентности (**183**, **470**) – позволяют распараллелить внутренний цикл
- Зависимости по памяти (массивы)
  - Никогда не зависят – нет проблем
  - Всегда – аналогично скалярам
  - Неизвестно – динамическая проверка

четные и **нечетные** итерации

- Расширяем на массивы понятия *Private*, *First-*, *Last- Private*, *Reduction* переменных
- Используем системы диофантовых уравнения и неравенств

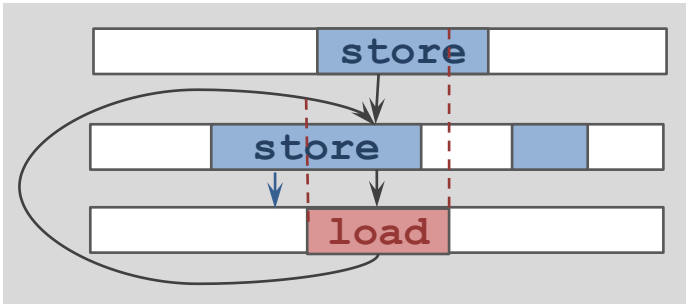
- Настройка эвристик; например, inline (**168.wupwise**) – много вызовов больших функций ~1850 инструкций; пропагация констант, клонирование, получаем 260 и 305
- Предварительные трансформации; code replication (**464.h264ref**) – из цикла получается 3, 2 их которых распараллеливаются

Private, First-, Last- private, Reduction

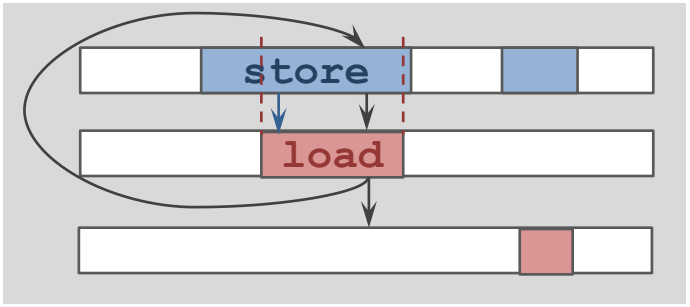


Распространим определения со скаляров на агрегатные объекты

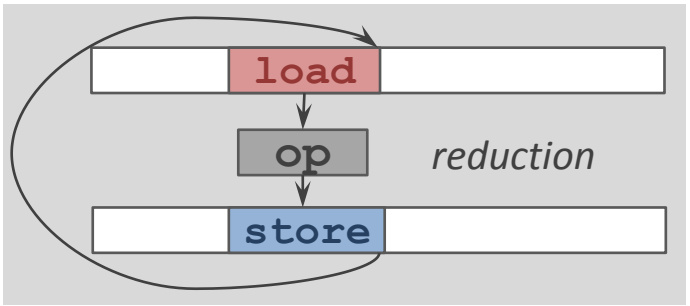
- **Private массив**: на каждой итерации LOAD читает из массива только то, что было записано на этой же итерации



- **First-private**: LOAD читает только либо то, что было записано на этой же итерации, либо то, что записано в массив до цикла



- **Last private**: private или first-private массив в чтением после цикла ТОЛЬКО того, что записано на последней итерации (*сильное – **достаточное** – условие*)



- **Reduction**: Аналогично скалярному случаю

Определение приватности

Пробуем доказать, что массив **`double a[]`** является private

Ставим решателю задачу в виде системы Диофантовых уравнений и неравенств

В качестве решателя **можно** использовать библиотеку Microsoft **z3**

```
for(i=0;i<n;i++){
  if(P(i))
    for(j=j1;j<j2;j++)
      a[j]=... ;
  for(k=k1;k<k2;k++)
    if(Q(k))
      ...=a[k];
}
```

$$\forall i\ \forall k: k_1 \le k \le k_1, Q(k), 0 \le i < n$$
$$\exists\, j: a + 8j = a + 8k, j_1 \le j < j_2, P(i), 0 \le i < n$$

Аллокация Копий Массивов

Для создания копии массивов надо знать их размер

1. В идеале – **найти диапазон адресов**, используемых в цикле (учесть предикаты, например, над циклом)
2. Использовать **информацию о размере** при аллокации исходного массива; возможно для массивов
   a. выделенных в той же процедуре, что и цикл, или
   b. глобальных.

   размер массива может быть больше используемой части, что увеличивает накладные расходы на инициализацию и восстановление

   для private-массивов достаточно оценки сверху

   для first- last- private и reduction – точный размер
3. Аллокация небольшого **массива с реаллокацией** и копированием при необходимости

   Небольшие накладные расходы; невозможно точно оценить затраты на перестроение
4. Вставка предварительного цикла определения диапазона используемых адресов

   существенные накладные расходы; точный размер до выделения копии массива

```
for(i=Lo;i<Hi;i++)
  a[5*i-7]
```

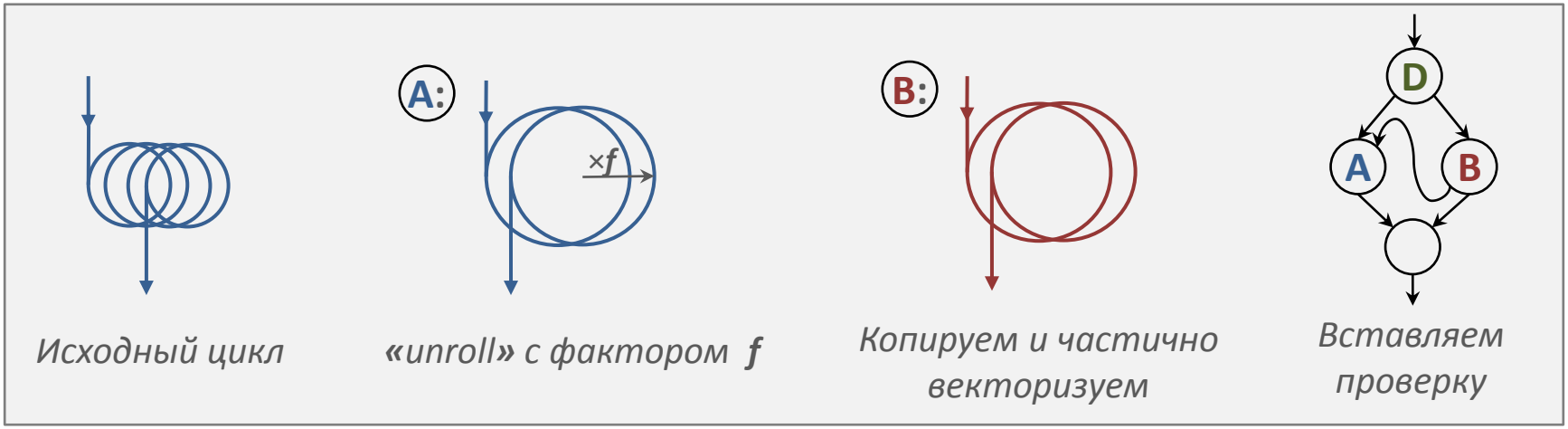
```
p = malloc(100500);
...
for(;;){
  p[]
```

```
arCpy = malloc(sz);
...
if(ind>sz){
 sz+=1050;
 realloc(arCpy,sz);
 // copy, if needed
```

```
for(;;)
  sz=max(sz,ind);
malloc(sz);
```

Векторизация

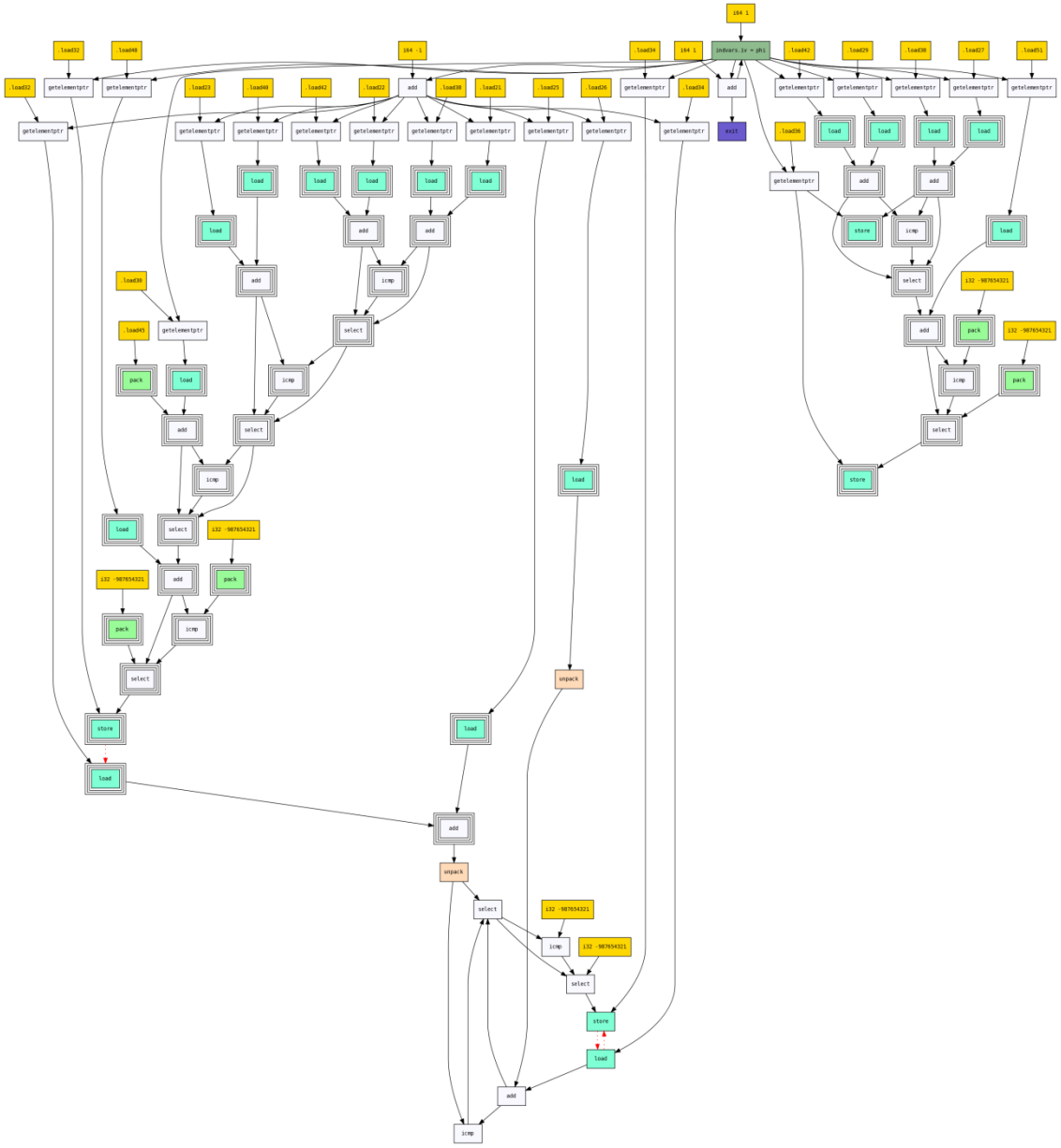
- Мелкозернистый параллелизм
- Основан на использовании SIMD-инструкций
- Рассматривается только цикловая векторизация
- Классическая схема: или векторизуются все инструкции цикла, или никакие
- Частичная векторизация: векторизуется часть инструкций цикла
- Подход UTL – комбинированная схема:
  - создается копия раскрученного цикла
  - к которой применяется частичная векторизация
  - оригинал держится невекторизованным
  - перед циклами вставляется динамическая проверка для выбора версии цикла



Векторизация 456.hmmer

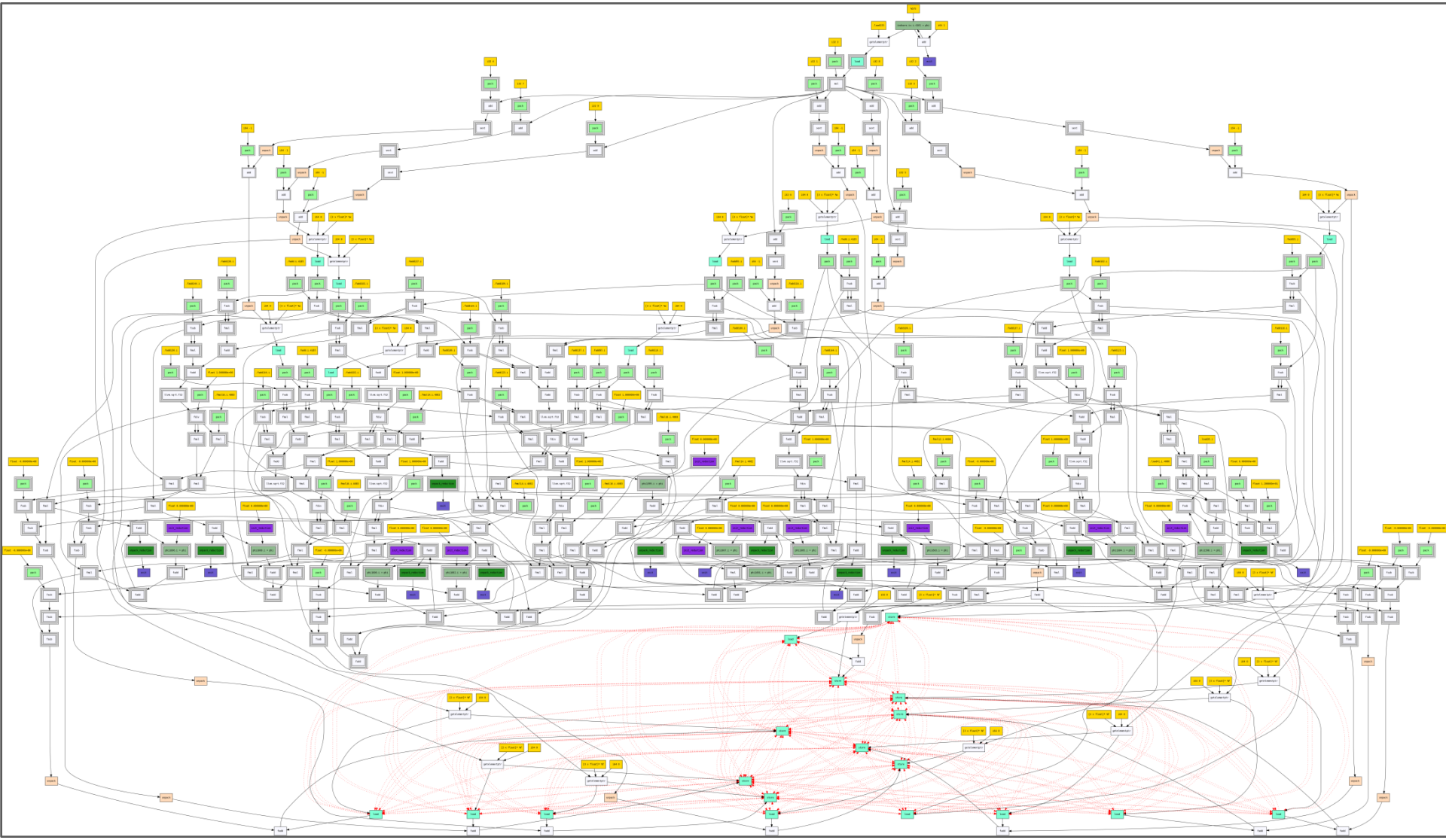
Показан граф зависимости исходного цикла

- черные стрелки – зависимость по регистрам
- красные – по памяти
- число рамочек – ранг вектора
- желтый – инварианты
- фиолетовый – выход
- зеленый – узлы запаковки
- розовый – узлы распаковки
- аквамарин – память
- x86: **87.3%**
- ARM: **16%**
- MIPS: **почти ничего**



Векторизация 435.gromacs

x86: **85.8%** ARM: **26%** MIPS: **ничего**



SMCC – Smart Compiler

- Мы интегрировали UTL с цепочкой LLVM – компилятор Clang
- Для FORTRAN – пришлось поддерживать
- **Benchmarks**: SPEC/CPU2006 test suit was run
- **Platforms**:

  ARM

  APM X-Gene XC-1 Mustang board Aarch64 **Octa** 1.6GHz, 8Gb RAM

  MIPS

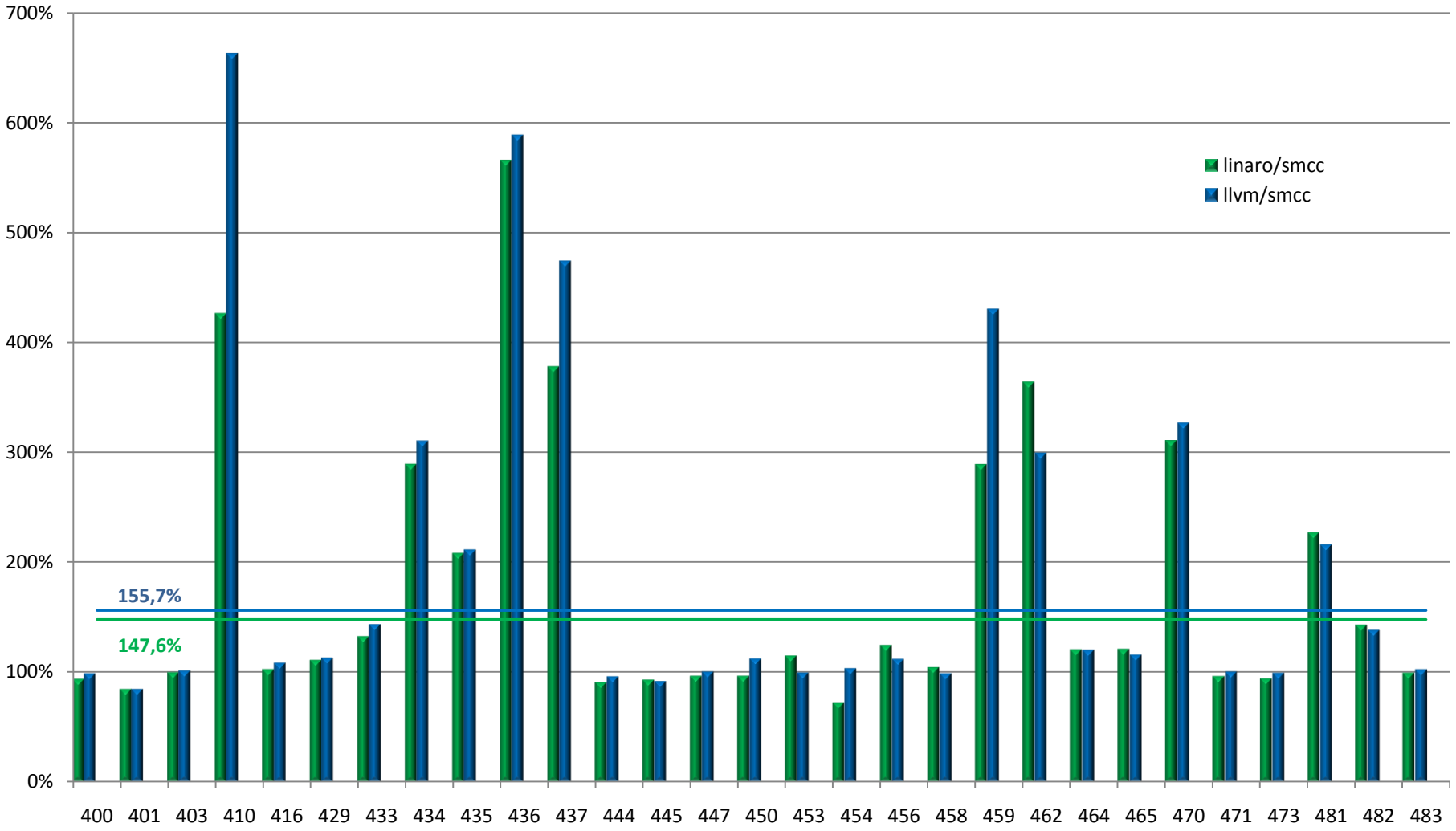
  Creator CI20 SoC: Ingenic JZ4780 CPU: **Dual** 1.2GHz XBurst MIPS32 little endian 32kI + 32kD per core 512K shared L2, 1Gbyte DDR3
- **Compilers for comparison**:
  - Linaro-GCC: 4.9 (for ARM) and GCC 4.9.2 (for MIPS)
  - Clang: clang-3.7 + dragonegg *(for FORTRAN bmks)*
- **Compiler options**: *(common for all compilers)*

```
-flto -O3 -mcpu=cortex-a57
```

- **Environment**: `OMP_PROC_BIND=true` is set to minimize migrations from core to core

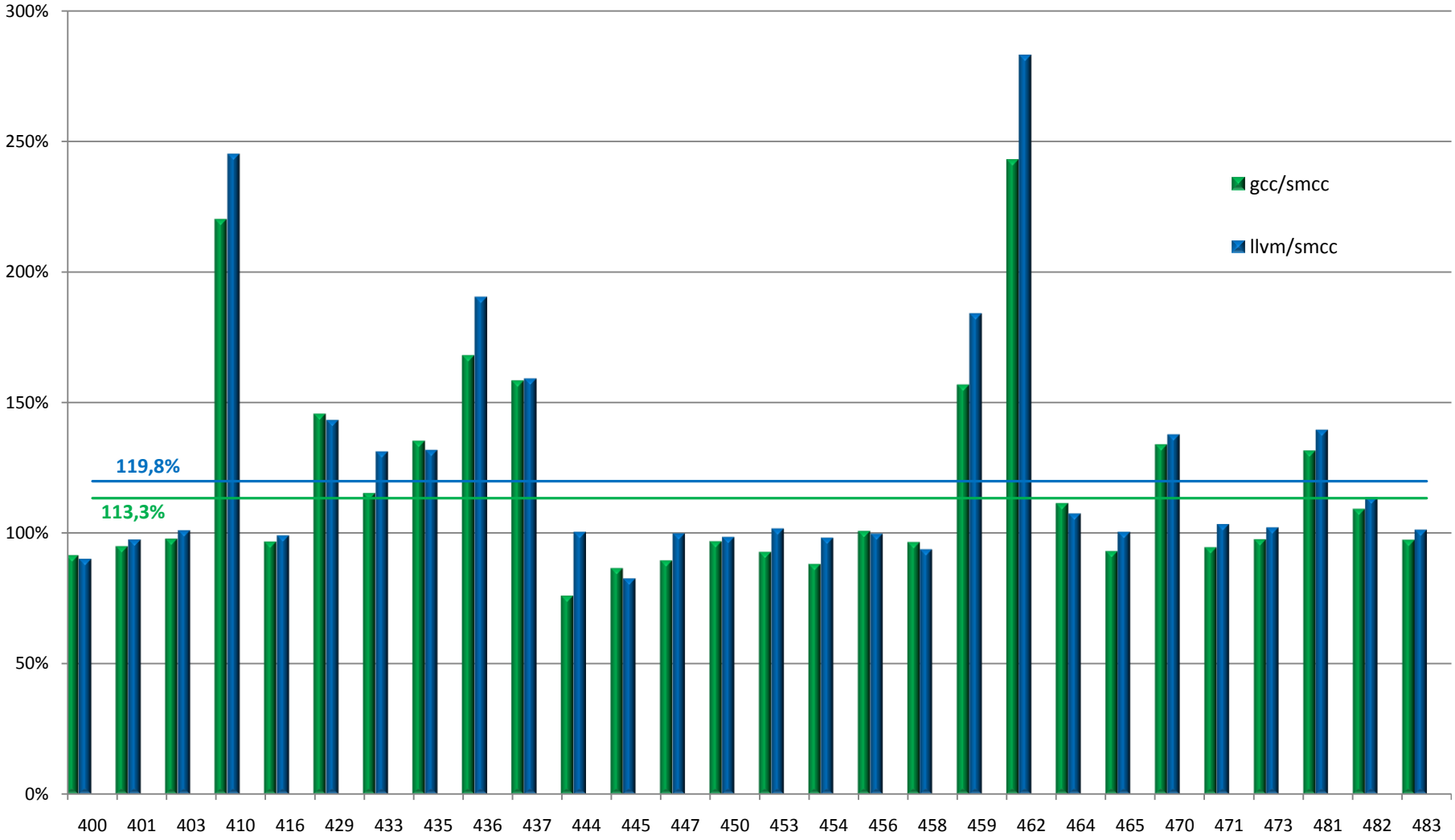
Производительность на ARM 8-core

- SPEC/CPU2006, reference data
- Smcc wins **47%** and **55%** Linaro-gcc and LLVM correspondingly (in geomean)



Производительность на MIPS 2-core

- SPEC/CPU2006 excluding 434.zeusmp, train data considered
- Smcc wins **13%** and **20%** GCC and LLVM



Производительность на ОДНОМ Ядре

- Авто-распараллеливание – основной драйвер прироста производительности: **50-550%** на 8-core ARM
- Но даже на одном ядре SMCC выигрывает у GCC и LLVM
  - **3%** и **5%** на ARM
  - **2.5%** и **6%** на MIPS соответственно
- На задаче **410.bwaves** на MIPS GCC и LLVM проигрывают **30%** и **44%**
- `pow(x, 1.75)` не был заменен на **`SQRT`** и **`MUL`** инструкции:

$$x^{1,75} = x \cdot \sqrt{x} \cdot \sqrt{\sqrt{x}};$$

- И GCC, и LLVM делают это для других архитектур

Создавать архитектуру без эффективных средств разработки может свести к нулю все усилия разработчиков железа