Как приблизиться с пиковому значению Flops/s? Сравнение архитектур x86 и ARMv8*

В.П. Никольский[1,2], В.В. Стегайлов[1,2]

Национальный исследовательский университет «Высшая школа экономики»[1],
Объединенный институт высоких температур РАН[2]

В данной работе мы сравниваем возможности компиляторов по построению векторизованного кода для современных процессоров ARMv8 и x86_64 и обсуждаем ассемблерный алгоритм достижения максимальной производительности на обеих архитектурах. Представлен высокооптимизированный микротест, достигающий максимальную долю от пиковой производительности на процессоре ARMv8, в то время как аналогичные программы для архитектуры x86_64 уже разработаны. Тесты были проведены на системе-на-чипе Nvidia Tegra X1 с четырьмя ядрами ARMv8 Cortex-A57.

Ключевые слова: тест, векторизация, пиковая производительность, компилятор

1. Введение

Современные вычислительные системы устроены очень сложно [1, 2]. На протяжении многих лет в области высокопроизводительных вычислений лидировали процессоры архитектуры x86, кроме того, они же использовались в настольных компьютерах и для преподавания программирования и архитектуры компьютеров. Таким образом, об этих процессорах накопилось довольно большое количество доступной информации.

За последние годы процессоры ARM, доминирующие на рынке мобильной и встраиваемой техники, значительно усилили своё присутствие среди высокопроизводительных систем [3, 4] и в суперкомпьютерной индустрии. С появлением 64-битного поколения ARMv8 в 2012 году [5] и соответствующих инструментов компиляции большую часть прикладных и научных программ можно с малыми усилиями перенести на эту платформу [6]. В 2016 году было анонсировано архитектурное расширение «Scalable Vector Extension», ориентированное на высокопроизводительные вычисления. К 2020–2022 году японской компанией Fujitsu запланировано завершение строительства суперкомпьютера экзафлопсного класса Post-K [7] на базе процессоров ARM. Однако относительно малое количество исследований этих процессоров, а порой и отсутствие достаточно качественной документации, затрудняет их оценку в общем контексте.

В предыдущих исследованиях процессоров ARM [8–11] мы использовали подход «сверху–вниз»: изначально не ставя себе цели разобраться во всех тонкостях работы процессора на уровне отдельных инструкций, мы использовали реальные научные приложения и тесты на языках высокого уровня, чтобы оценить свойства и эффективность связки аппаратного и программного обеспечения (в том числе энергоэффективность), лишь иногда прибегая к низкоуровневому анализу для объяснения наблюдаемых результатов. Мотивацией для данной работы послужила критическая масса вопросов, ответы на которые кроются именно в преобразовании кода компилятором и нюансах исполнения этого кода на ядрах процессора. В искусственном примере достижения максимальной производительности мы ставим цель продемонстрировать, какие механизмы ускорения рассчётов заложены в современных процессорах ARM и Intel, как эти механизмы задействовать, а также сравнить их. Достижение максимальной производительности за счет глубокой оптимизации кода — одно из магистральных направлений развития современных высокопроизводительных вычислительных

*Исследование финансировалось в рамках государственной поддержки ведущих университетов Российской Федерации «5-100».

технологий [12]. Чисто эмпирическая работа, направленная на определение пиковой производительности процессоров предыдущего поколения ARMv7 (Cortex-A15 и слабее) опубликована в 2013 году [13].

В рамках данной работы мы сперва рассмотрим «экосистему» вокруг архитектур ARMv8 и x86_64, далее в 3 части опишем устройства, на которых запускались тесты. В 4 части будет рассчитана теоретическая производительность процессоров с описанием работы конвейера процессора ARM Cortex-A57. Затем будут представлены тесты, измеряющие производительность различных элементарных операций, а их результаты будут проанализированы с учётом имеющегося опыта. Части 6.2 и 6.1 посвящены популярным тестам, в том числе на языке высокого уровня.

2. Обзор инструментов

Для анализа производительности прикладных программ на платформе x86_64 существует набор инструментов, доступных по академической лицензии. Они включают в себя компилятор, отладчик, профилировщик и специальные средства, входящие в состав Intel Parallel Studio. Эта программа позволяет для каждого цикла оценить уровень векторизации, показывает, какой набор инструкций был задействован при векторизации, а также предлагает способы улучшения результатов.

Что касается ARM, то в данной работе были использованы компилятор и кросс-компилятор gcc, отладчик gdb и дизассемблирование с помощью objdump. ARM активно поддерживает разработку gcc и OpenMP для своих процессоров, но разработку коммерческого компилятора они базируют на LLVM [14] (в декабре 2016 года анонсирована первая версия). Кроме того ARM выпустили высокопроизводительные математические библиотеки ARM Performance Libraries, включающие алгоритмы BLAS, LAPACK и FFT со стандартными интерфейсами.

В апреле 2017 года ARM проводит бета-тестирование новых инструментов профилировки, параллельной отладки и оценки производительности. Таким образом, в течение последнего год ARM активно создает среду коммерческих инструментов для разработки высокопроизводительных параллельных приложений, очень похожую на ту, которую разрабатывает Intel.

3. Аппаратное и программное обеспечение

В данной работе тесты запускались на двух устройствах: ноутбуке с процессором Intel и системе-на-чипе Nvidia Jetson TX1, подробные исследования свойств которой уже были нами опубликованы [10, 11].

3.1. Nvidia Jetson TX1

Jetson TX1 основан на 64-битной системе-на-чипе [15] Tegra X1 с памятью LPDDR4 (1600 МГц). Система включает 4 ядра Cortex-A57, работающие на частоте до 1.9 GHz, 4 более слабых ядра Cortex-A53 в конфигурации big.LITTLE, подменяющие основные ядра при низкой нагрузке и два SM графического ускорителя Maxwell, работающего на частоте 988 ГГц и содержащего 256 ядер CUDA. Каждое ядро Cortex-A57 имеет L1 кеш объемом 48 Кбайт, 32 Кбайт кеша данных уровня L1. Также присутствует L2 кеш, 2 Мбайт которого распределены между всеми ядрами. Система имеет 4 ГБ памяти.

В качестве операционной системы установлен Linux Ubuntu 16.01 LTS с 64-битным aarch64 ядром. Используется 64-битный набор инструментов компиляции gcc 5.4.0 Ubuntu/Linaro и 64-битное программное окружение.

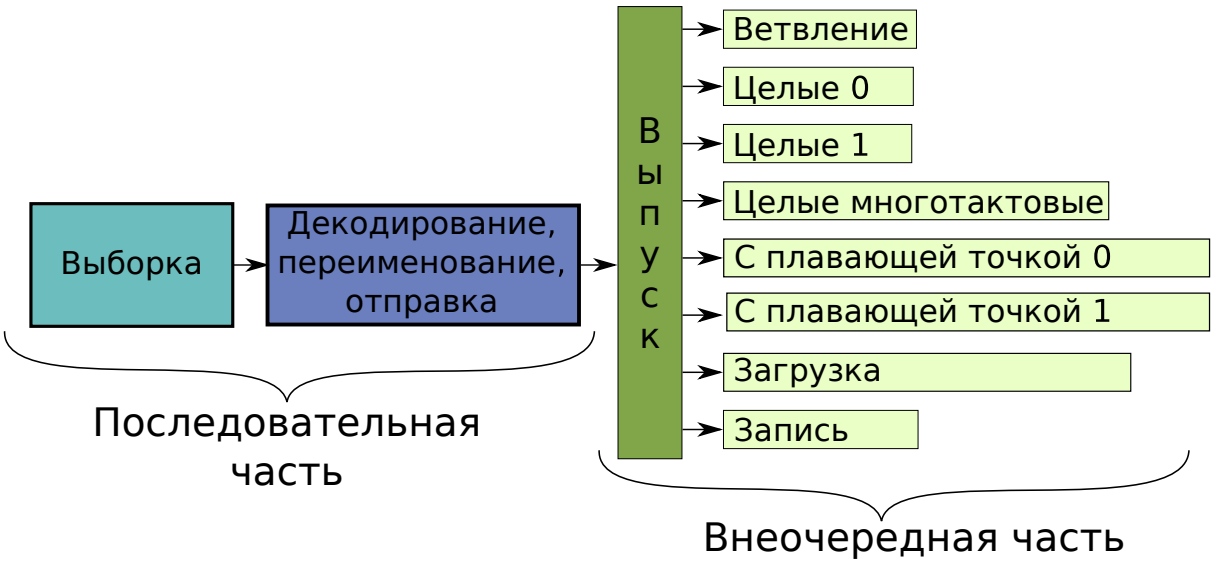


Рис. 1: Схема конвейера ARM

3.2. Intel Haswell

В качестве представителя x86_64 мы использовали Intel Core(TM) i5-4210H CPU @ 2.90 GHz с 8 ГБ памяти. Процессор имеет архитектуру Haswell [16].

В качестве операционной системы установлен Linux Ubuntu 16.01.02 LTS с 64-битным x86_64 ядром. В качестве компилятора использован gcc 5.4.0.

4. Теоретическая производительность

4.1. ARMv8

Ядра Cortex-A57 [8,9] имееют глубокий суперскалярный конвейер [17] (высокоуровневая схема представлена на рис. 1), содержащий внеочередную (out-of-order) и последовательную (in-order) части. Последовательная часть включает в себя выборку инструкций, их декодирование во внутренние микрооперации, переименование и отправку микроопераций. После выпуска микрооперации попадают на исполнение во внеочередные вычислительные конвейеры — для нас особый интерес представляет наличие двух конвейеров обработки чисел с плавающей точкой и векторных операций (FP/ASIMD). Последовательная часть может обрабатывать до трех микроопераций за такт, в том числе может за один такт отправить две микроинструкции в разные FP/ASIMD конвейеры.

Каждая инструкция характеризуется задержкой (latency) и пропускной способностью (throughtput). Эта информация доступна не для всех процессоров, но для серии Cortex-A57 эти значения опубликованы [17].

Определение 1.
Задержка — это количество тактов, после которых записываемые инструкцией данные доступны для следующей операции.

Определение 2.
Пропусная способность — (здесь) максимальное количество инструкций в такт, которые может быть достигнуто для данной группы инструкций на одном ядре процессора. Иногда (например в Intel Intrinsics guide) используется обратное определение: количество тактов, требуемое для выполнения инструкции.

Задержка препятствует непрерывной загрузке конвейера, а пропускная способность ограничивает сверху пиковую производительность. Отдельно отметим, что для научных и

инженерных расчётов интерес в первую очередь представляет производительность в смысле операций над числами с плавающей точкой двойной точности (Float64) в секунду. В наборе инструкций ARMv8 содержатся векторные операции, которые в одной инструкции кодируют и выполняют параллельно сразу несколько арифметических действий над несколькими Float64 операндами. Таким образом, с учётом полной загрузки конвейера производительность одного ядра процессора в смысле Flops/s на данной операции может быть рассчитана следующим образом:

$$R(op) = CPU_{freq} \cdot k_{OP}(op) \cdot k_{SIMD}(op) \cdot troughtput(op),$$

где CPU_{freq} – тактовая частота ядра, k_{OP} — количество арифметических операций, совершаемых над каждым операндом, k_{SIMD} — количество операндов, $troughtput$ — пропускная способность операции. Пиковая производительность определяется при выборе такой операции или последовательности операций, что множитель при CPU_{freq} максимален

$$R_{peak} = \max_{op} R(op).$$

Для Cortex-A57 пиковую производительность обеспечивает группа операторов `FMLA` — fused multiply accumulate. Эта операция имеет форму $d = d * m + a$, причем между операцией умножения и сложения не происходит дополнительного округления. Она имеет большую задержку 9 циклов, но крайне высокую пропускную способность 1, а также имеет векторную форму, в которой принимает 2 оператора двойной точности Float64, или 4 оператора одинарной точности Float32. Таким образом, теоретическая производительность на Cortex-A57 в составе SoC Jetson TX1 с тактовой частотой 1.9 ГГц составляет: $R(FMLA) = 1.9(GHz) \cdot k_{OP}(FMLA) \cdot k_{SIMD}(FMLA) \cdot troughtput(FMLA) = 1.9(GHz) \cdot 2 \cdot 2 \cdot 1(Flop) = 7.6\ GFlop/s$.

4.2. x86_64 AVX2

Вопрос достижения пиковой производительности на процессорах архитектуры x86_64 в целом и Intel Haswell с набором инструкций AVX2 в частности хорошо изучен [18]. Для Float64 можно получить 16 Flop на такт при выполнении двух fused multiply-add инструкций над четырьмя операндами каждой ($R(FMA) = CPU_{freq} \cdot 2 \cdot 4 \cdot 2 = 16 \cdot CPU_{freq}$).

5. Разработка тестов

5.1. Описание

Был написан набор тестов (микробенчмакров) на базе кода Mysticial/Flops, который доступен на GitHub и представляет из себя бенчмарки для процессоров x86_64 (автор Alexander Yee). Новые программы тестируют разные последовательности элементарных арифметических операций на быстродействие для того, чтобы выявить максимальную достижимую производительность.

Каждый тест заключается в многократном повторении последовательности элементарных арифметических команд с измерением времени выполнения. Это позволяет точно оценить время выполнения каждой отдельной команды, или, что то же самое, количество операций, выполняемых за один такт процессора. На основании этой информации можно делать выводы о том, какие механизмы увеличения производительности можно задействовать в том или ином случае.

Бенчмарк был создан с использованием intrinsic функций компилятора gcc — это функции, которые непосредственно представляют ассемблерные команды для использования в коде высокоуровневого языка без необходимости прибегать к использованию ассемблерных вставок. Тем не менее, такие функции специфичны для разных наборов процессорных инструкций, так как intrinsic функции находятся в отношении эквивалентности с инструкциями процессора. В простейшем случае для адаптации кода для новой процессорной

архитектуры достаточно было подключить соответствующие заголовочные файлы и изменить intrinsic функции в коде. В случае архитектуры ARMv8 было необходимо подготовить данные-операнды нужного типа, например, для использования векторных инструкций. Для экспериментов с нетипичными последовательностями инструкций были самостоятельно разработаны отдельные вычислительные ядра бенчмарка.

Важно учесть явление задержки (см. опр. 1), иначе зависимость соседних инструкций по данным приведет к конфликтам конвейера и его остановке. Для того, чтобы этого избежать, вручную создаются достаточно длинные последовательности взаимно независимых по записи инструкций (рис. 2). Таким образом, можно утверждать, что разработанный тест позволяет достичь максимальной производительности, так как он реализует последовательность инструкций, обеспечивающую максимальную загруженность системы [1] (что объясняется в разделе 4.1), а также стремится уменьшить влияние накладных расходов. Тем не менее, к 100% пиковой производительности можно только приблизиться, так как существуют такие препятствия, как «разгон конвейера», конечное число регистров и другие.

Тесты могут быть запущены на одном или нескольких ядрах процессора с помощью технологии OpenMP.

```
const float64x2_t mul0 = {TEST_MUL_MUL,TEST_MUL_MUL};
const float64x2_t mul1 = {TEST_MUL_DIV,TEST_MUL_DIV};
const float64x2_t add0 = {TEST_ADD_ADD,TEST_ADD_ADD};
for (size_t i = 0; i < iterations; ++i){
    r0 = vfmaq_f64(r0, mul0, add0); r1 = vfmaq_f64(r1, mul0, add0);
    r2 = vfmaq_f64(r2, mul0, add0); r3 = vfmaq_f64(r3, mul0, add0);
    r4 = vfmaq_f64(r4, mul0, add0); r5 = vfmaq_f64(r5, mul0, add0);
    r6 = vfmaq_f64(r6, mul0, add0); r7 = vfmaq_f64(r7, mul0, add0);
    r8 = vfmaq_f64(r8, mul0, add0); r9 = vfmaq_f64(r9, mul0, add0);
    rA = vfmaq_f64(rA, mul0, add0); rB = vfmaq_f64(rB, mul0, add0);
    r0 = vfmsq_f64(r0, mul1, add0); r1 = vfmsq_f64(r1, mul1, add0);
    r2 = vfmsq_f64(r2, mul1, add0); r3 = vfmsq_f64(r3, mul1, add0);
    r4 = vfmsq_f64(r4, mul1, add0); r5 = vfmsq_f64(r5, mul1, add0);
    r6 = vfmsq_f64(r6, mul1, add0); r7 = vfmsq_f64(r7, mul1, add0);
    r8 = vfmsq_f64(r8, mul1, add0); r9 = vfmsq_f64(r9, mul1, add0);
    rA = vfmsq_f64(rA, mul1, add0); rB = vfmsq_f64(rB, mul1, add0);
}
```

Рис. 2: Код C++ вычислительного ядра, доставляющего максимальную производительность на Cortex-A57

5.2. Компиляция

Компиляция осуществляется gcc с флагами `'-g -O2 -std=c++0x -static -fopenmp'`. После компиляции полученный исполняемый файл обязательно нужно верифицировать посредством отладки и дизассемблирования (см. рис. 3), поскольку при компиляции даже без подключения опасных оптимизаций вполне вероятны ситуации, приводящие к некорректным результатам. В случае, если компилятор исключит из кода выполнение части инструкций, производительность окажется завышена, причем числовой результат выполнения арифметических операций может быть с высокой точностью сохранён.

6. Обсуждение результатов

На рис. 5 показана производительность, достигаемая при использовании разных инструкций на рассматриваемом оборудовании. В полном соответствии с теорией максимальные значения достигаются при использовании инструкций типа fused multiply add. На Cortex-A57 в нашем тесте при загрузке одного ядра было достигнуто значение 91.2% от теоретической производительности, а на Intel Haswell 90.6%. На рис. 6 показано, что специально оптимизированный тест показывает по сравнению с универсальными тестами лучшие результаты.

```
                        ...
4058e4:     4e60cc15    fmla    v21.2d, v0.2d, v0.2d
4058e8:     eb03003f    cmp     x1, x3
4058ec:     4e60cc07    fmla    v7.2d, v0.2d, v0.2d
4058f0:     4e60cc10    fmla    v16.2d, v0.2d, v0.2d
4058f4:     4e60cc04    fmla    v4.2d, v0.2d, v0.2d
4058f8:     4e60cc02    fmla    v2.2d, v0.2d, v0.2d
4058fc:     4e60cc13    fmla    v19.2d, v0.2d, v0.2d
405900:     4e60cc14    fmla    v20.2d, v0.2d, v0.2d
405904:     4e60cc05    fmla    v5.2d, v0.2d, v0.2d
405908:     4e60cc06    fmla    v6.2d, v0.2d, v0.2d
40590c:     4e60cc03    fmla    v3.2d, v0.2d, v0.2d
405910:     4e60cc11    fmla    v17.2d, v0.2d, v0.2d
405914:     4e60cc12    fmla    v18.2d, v0.2d, v0.2d
405918:     4e60cc35    fmla    v21.2d, v1.2d, v0.2d
40591c:     4e60cc27    fmla    v7.2d, v1.2d, v0.2d
405920:     4e60cc30    fmla    v16.2d, v1.2d, v0.2d
405924:     4e60cc24    fmla    v4.2d, v1.2d, v0.2d
405928:     4e60cc22    fmla    v2.2d, v1.2d, v0.2d
40592c:     4e60cc33    fmla    v19.2d, v1.2d, v0.2d
405930:     4e60cc34    fmla    v20.2d, v1.2d, v0.2d
405934:     4e60cc25    fmla    v5.2d, v1.2d, v0.2d
405938:     4e60cc26    fmla    v6.2d, v1.2d, v0.2d
40593c:     4e60cc23    fmla    v3.2d, v1.2d, v0.2d
405940:     4e60cc31    fmla    v17.2d, v1.2d, v0.2d
405944:     4e60cc32    fmla    v18.2d, v1.2d, v0.2d
405948:     54fffcc1    b.ne    4058e0 <function>
                        ...
                        (a)
```

```
                        ...
4053f0:     6e60dce7    fmul    v7.2d, v7.2d, v0.2d
4053f4:     eb00003f    cmp     x1, x0
4053f8:     4e60d463    fadd    v3.2d, v3.2d, v0.2d
4053fc:     6e60dcc6    fmul    v6.2d, v6.2d, v0.2d
405400:     4e60d4a5    fadd    v5.2d, v5.2d, v0.2d
405404:     6e60dc84    fmul    v4.2d, v4.2d, v0.2d
405408:     4e60d610    fadd    v16.2d, v16.2d, v0.2d
40540c:     6e60de94    fmul    v20.2d, v20.2d, v0.2d
405410:     4e60d79c    fadd    v28.2d, v28.2d, v0.2d
405414:     6e60df39    fmul    v25.2d, v25.2d, v0.2d
405418:     4e60d7de    fadd    v30.2d, v30.2d, v0.2d
40541c:     6e60df18    fmul    v24.2d, v24.2d, v0.2d
405420:     4e60d7bd    fadd    v29.2d, v29.2d, v0.2d
405424:     6e62def7    fmul    v23.2d, v23.2d, v2.2d
405428:     4e60d54a    fadd    v10.2d, v10.2d, v0.2d
40542c:     6e62ded6    fmul    v22.2d, v22.2d, v2.2d
405430:     4e60d673    fadd    v19.2d, v19.2d, v0.2d
405434:     6e62de52    fmul    v18.2d, v18.2d, v2.2d
405438:     4e60d529    fadd    v9.2d, v9.2d, v0.2d
40543c:     6e62deb5    fmul    v21.2d, v21.2d, v2.2d
405440:     4e60d7ff    fadd    v31.2d, v31.2d, v0.2d
405444:     6e62df7b    fmul    v27.2d, v27.2d, v2.2d
405448:     4e60d631    fadd    v17.2d, v17.2d, v0.2d
40544c:     6e62df5a    fmul    v26.2d, v26.2d, v2.2d
405450:     4e60d508    fadd    v8.2d, v8.2d, v0.2d
405454:     6e60dce7    fmul    v7.2d, v7.2d, v0.2d
405458:     4e61d463    fadd    v3.2d, v3.2d, v1.2d
40545c:     6e60dcc6    fmul    v6.2d, v6.2d, v0.2d
405460:     4e61d4a5    fadd    v5.2d, v5.2d, v1.2d
405464:     6e60dc84    fmul    v4.2d, v4.2d, v0.2d
405468:     4e61d610    fadd    v16.2d, v16.2d, v1.2d
40546c:     6e60de94    fmul    v20.2d, v20.2d, v0.2d
405470:     4e61d79c    fadd    v28.2d, v28.2d, v1.2d
405474:     6e60df39    fmul    v25.2d, v25.2d, v0.2d
405478:     4e61d7de    fadd    v30.2d, v30.2d, v1.2d
40547c:     6e60df18    fmul    v24.2d, v24.2d, v0.2d
405480:     4e61d7bd    fadd    v29.2d, v29.2d, v1.2d
405484:     6e62def7    fmul    v23.2d, v23.2d, v2.2d
405488:     4e61d54a    fadd    v10.2d, v10.2d, v1.2d
40548c:     6e62ded6    fmul    v22.2d, v22.2d, v2.2d
405490:     4e61d673    fadd    v19.2d, v19.2d, v1.2d
405494:     6e62de52    fmul    v18.2d, v18.2d, v2.2d
405498:     4e61d529    fadd    v9.2d, v9.2d, v1.2d
40549c:     6e62deb5    fmul    v21.2d, v21.2d, v2.2d
4054a0:     4e61d7ff    fadd    v31.2d, v31.2d, v1.2d
4054a4:     6e62df7b    fmul    v27.2d, v27.2d, v2.2d
4054a8:     4e61d631    fadd    v17.2d, v17.2d, v1.2d
4054ac:     6e62df5a    fmul    v26.2d, v26.2d, v2.2d
4054b0:     4e61d508    fadd    v8.2d, v8.2d, v1.2d
4054b4:     54fff9c1    b.ne    4053ec <function>
                        ...
                        (6)
```

Рис. 3: Ассемблерный листинг вычислительного ядра, обеспечивающего максимальную производительность на Cortex-A57 (а) в сравнении с кодом, выполняющим близкие по математическому смыслу функции менее эффективно (б)

Стоит отметить, что fused multiply add операции используются далеко не в каждом алгоритме, а тем более не в каждом алгоритме есть благоприятные условия для наиболее эффективного использования этих инструкций без остановок конвейера. Тем не менее, эти инструкции могут быть крайне эффективны например в некоторых матричных алгоритмах и при обработке сигналов. Более универсальные сложение и умножение обеспечивают примерно вдвое меньшую производительность.

Intel FMA3 и Aarch64 различаются в видах инструкций FMA. В наборе инструкций Aarch64 набор инструкций значительно меньше и проще. Инструкции кодируют только простейшие последовательности арифметических действий над векторами, такие как «умножить и прибавить» или «умножить и вычесть», в то время как в FMA3 содержатся например операции, выполняющие разные действия для чётных и нечётных элементов вектора, причем с сохранением быстродействия. Кроме того, каждая FMA операция в наборе FMA3 представлена в трёх формах, в которых три операнда занимают разные места в арифметическом выражении (что отображается в мнемониках, как 132, 213 и 231). По всей видимости, столь существенное различие в разнообразии предоставляемых инструкций является следствием принадлежности процессоров к разным типам процессорных архитектур — complex instruction set computer (CISC) и reduced instruction set computer (RISC), и использования в наборе инструкций FMA3 специальной схемы кодирования VEX.

```
double c = 0.5;
for(i=0; i<N; ++i){
    double b = 0.8;
    b = b*A[i] + c;
    b = b*A[i] + c;
    b = b*A[i] + c;
    ...
    A[i] = b;
}
```

Рис. 4: Вычислительное ядро Empirical Roofline Toolkit

6.1. Сравнение с тестом ERT

Empirical Roofline Toolkit (ERT) [19] является новым универсальным инструментом тестирования производительности, кросс-платформенная реализация которого написана на языке C++. В качестве результата этот инструмент строит модель Roofline для вычислительной системы. Модель Roofline включает в себя оценку пиковой производительности и пропускной способности памяти, так что ERT ставит целью измерение обеих величин с максимальной точностью.

ERT мы широко использовали в предыдущих работах. Было показано, что ERT корректно определяет максимальную производительность для Intel Haswell (x86_64, AVX2). Его исходный код (рис. 4) не сводится непосредственно к матричным выражениям, но отлично ложится на набор инструкций Haswell FMA3 (через последовательность инструкций vfmadd132sd). С другой стороны, в то время как он заявлен универсальным тестом, не ориентированным специально на архитектуру x86_64, при компиляции ERT на ARMv8 наилучшим способом мы наблюдаем значительную недооценку пиковой производительности.

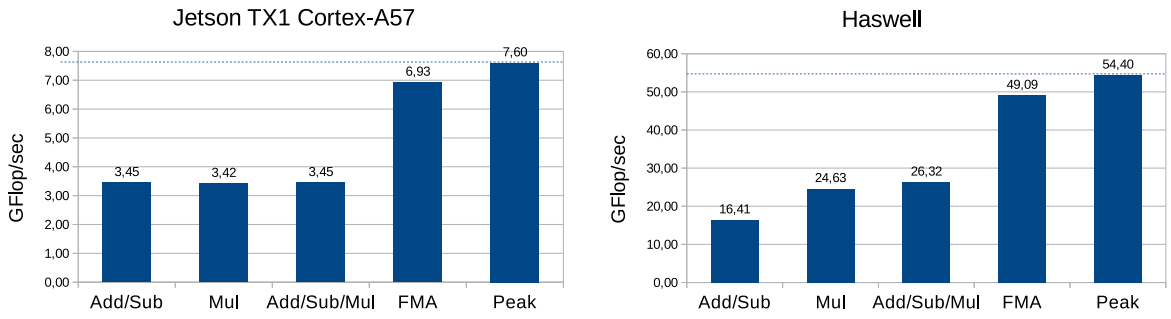
6.2. Сравнение с тестом LINPACK

High Performance LINPACK (HPL) является стандартным инструментом для тестирования максимальной реально достижимой производительности. Вычислительной нагрузкой в нём выступает решение случайно сгенерированной плотной системы линейных алгебраических уравнений в арифметике двойной точности на машинах с распределённой памятью. Параллелизация в этой задаче реализуется с помощью библиотек Message Passing Interface (MPI), за численные алгоритмы линейной алгебры отвечает Basic Linear Algebra Subprograms (BLAS) или Vector Signal Image Processing Library (VSIPL). Указанные библиотеки являются типичными компонентами высокопроизводительных приложений, и поэтому имеют множество оптимизированных реализаций, в том числе Intel MKL и ARM Performance Library, которые были использованы в этой работе. Для процессоров Intel публикуется реализация бенчмарка Intel Distribution for LINPACK Benchmark.

Тест LINPACK показывает (рис. 6) довольно близкие результаты при запуске на ARMv8 и Intel Haswell. Это связано с хорошей оптимизацией библиотек линейной алгебры для всех архитектур, которые ориентированы на высокопроизводительные вычислений, в том числе x86_64 и ARMv8. Вообще, LINPACK является репрезентативным примером решения задачи линейной алгебры с использованием библиотек, используемых в реальных приложениях.

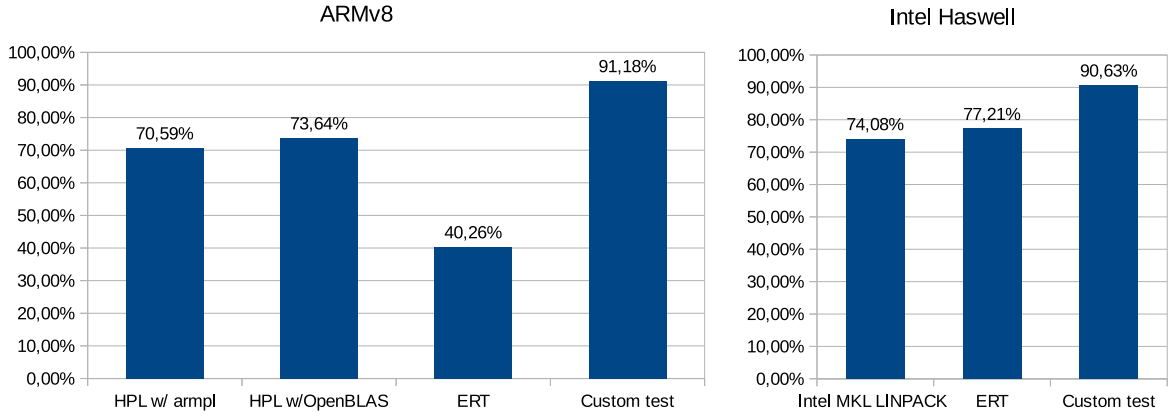
6.3. Производительность реальных приложений

Молекулярно-динамические расчёты составляют значительную долю всех суперкомпьютерных вычислений в мире. Наши предыдущие работы [8, 9] посвящены тестам молекулярно-динамического кода на процессорах ARMv8 и ARMv7. Было показано, что в зависимости от задачи производительность может быть ограничена как пиковой производительностьи процес-



(а) (б)

Рис. 5: Соотношение производительности, достижимой с использованием разных инструкций ARM Cortex-A57 (а) и Intel Haswell (б)



(а) (б)

Рис. 6: Соотношение производительности, достижимой с использованием разных тестов на Cortex-A57 (а) и Intel Haswell (б)

сора, так и пропускной способностью памяти [10, 11]. Специфика МД-кода такова, что в нём почти не задействуются операции FMA. Также он не поддаётся автовекторизации современными компиляторами, и даже при ручной векторизации её эффективность ограничена. Механизм радиуса обрезки потенциала, повсеместно используемый в молекулярно-динамических задачах, вносит в самый «тяжелый» вычислительный цикл программы условный переход. Всё это составляет специфичный характер нагрузки процессоров при МД-расчётах, которых значительно отличается от матричных вычислений, традиционно используемых для тестирования производительности процессоров. Таким образом, производительность процессоров на реальных задачах в общем случае не может быть прямо связана с результатами тестов пиковой производительности процессоров. Наши предыдущие публикации в том числе представляют собой попытку построить связь между оценками пиковой производительности процессоров ARM и x86 и их произвоидительностью на реальных молекулярно-динамических расчётах.

6.4. Энергоэффективность

В двух наших первых работах, посвященных процессорам ARM [8, 9], вопрос энергоэффективности новых процессоров на реальных молекулярно-динамических приложениях был одним из ключевых. В исследовании использовался младший из процессоров линейки Cortex-A, относящийся к устаревшей на данный момент (но ещё актуальной на момент исследования) архитектуре ARMv7. Были произведены точные замеры мощности системы-на-чипе на базе Cortex-A5 в реальных задачах. Были сделаны оценки, которые показали высокую энергоэффективность этого устройства по сравнению с серверами на базе процессоров Intel Ivy Bridge и Haswell. Исследование энергоэффективности процессоров ARM было продолжено следующей в работе [10] на более мощных процессорах Cortex-A15 (ARMv7) и Cortex-A57 (ARMv8). Было измерено энергопотребление систем в различных режимах.

Следует учитывать, что в исследованиях использовались компактные устройства, высокая энергоэффективность которых может быть во многом связана с низкой мощностью. Для того, чтобы ответить на этот вопрос, требуется провести замера энергопотребления серверного процессора ARM. В данный момент на рынке процессоры ARM в серверной конфигурации только начали появляться и пока не доступны в России.

7. Заключение

В данной работе была подробно рассмотрена работа вычислительного конвейера ядер Cortex-A57. Для процессоров с архитектурой Cortex-A57 была проанализирована теоретическая пиковая производительность с точки зрения операций над числами с плавающей точкой двойной точности. Был реализован соответствующий программный код, максимально реализующий возможности архитектуры ARMv8 по векторизации арифметических операций и по своей производительности приближающийся к теоретической пиковой производительности. С использованием процессора Nvidia Tegra X1 проведено сравнение максимальной производительности различных операций со значениями типа Float64 на архитектурах Cortex-A57 и Haswell. Полученные результаты сопоставляются с результатами классического теста LINPACK и нового теста на языке высокого уровня Empirical Roofline Toolkit. Показано, что несмотря на заявленную универсальность последнего, он показывает значительное несоответствие результатов действительности на ARMv8, в то время как для тестирования современных процессоров Intel он подходит отлично.

Литература

1. В. В. Воеводин и Вл В. Воеводин. *Параллельные вычисления.* БХВ-Петербург СПб, 2002.

2. David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

3. J. Goodacre and A. N. Sloss. Parallelism and the ARM instruction set architecture. *Computer*, 38(7):42–50, July 2005. doi:10.1109/MC.2005.239.

4. David D Pruitt and Eric A Freudenthal. Preliminary investigation of mobile system features potentially relevant to hpc. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*, E2SC '16, pages 54–60, Piscataway, NJ, USA, 2016. IEEE Press. doi:10.1109/E2SC.2016.13.

5. M. A. Laurenzano, A. Tiwari, A. Cauble-Chantrenne, A. Jundt, W. A. Ward, R. Campbell, and L. Carrington. Characterization and bottleneck analysis of a 64-bit ARMv8 platform. In

2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 36–45, April 2016. doi:10.1109/ISPASS.2016.7482072.

6. Ananta Tiwari, Kristopher Keipert, Adam Jundt, Joshua Peraza, Sarom S. Leang, Michael Laurenzano, Mark S. Gordon, and Laura Carrington. Performance and energy efficiency analysis of 64-bit arm using gamess. In *Proceedings of the 2Nd International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '15, pages 8:1–8:10, New York, NY, USA, 2015. ACM. doi:10.1145/2834899.2834905.

7. Yutaka Ishikawa. System software in Post K supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. ACM. URL: http://dl.acm.org/citation.cfm?id=2807591.2897785.

8. В. Никольский и В. Стегайлов Эффективность процессоров архитектуры ARM для расчетов классической молекулярной динамики *Вычислительные методы и программирование*, 16(4):578–585, 2015. URL: http://num-meth.srcc.msu.ru/english/zhurnal/tom_2015/v16r454.html.

9. V. Nikolskiy and V Stegailov. Floating-point performance of ARM cores and their efficiency in classical molecular dynamics. *Journal of Physics: Conference Series*, 681(1):012049, 2016. URL: http://stacks.iop.org/1742-6596/681/i=1/a=012049.

10. V. P. Nikolskiy, V. V. Stegailov, and V. S. Vecher. Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 682–689, July 2016. doi:10.1109/HPCSim.2016.7568401.

11. Vsevolod Nikolskii, Vyacheslav Vecher, and Vladimir Stegailov. Performance of MD-algorithms on hybrid systems-on-chip Nvidia Tegra K1 & X1. *Communications in Computer and Information Science*, pages 199–211, 2016. doi:10.1007/978-3-319-55669-7_16.

12. Alexander Heinecke, Wolfgang Eckhardt, Martin Horsch, and Hans-Joachim Bungartz. *Supercomputing for Molecular Dynamics Simulations*. SpringerBriefs in Computer Science. Springer International Publishing, 2015. doi:10.1007/978-3-319-17148-7.

13. Rahul Garg. *Exploring the Floating Point Performance of Modern ARM Processors*. AnandTech, 2013. URL: http://www.anandtech.com/show/6971/exploring-the-floating-point-performance-of-modern-arm-processors.

14. Darren Cepulis. ARM antes up for an HPC software stack. *The Next Platform*, March 2017. URL: https://www.nextplatform.com/2017/03/15/arm-antes-hpc-software-stack/.

15. Rochit Rajsuman. *System-on-a-Chip: Design and Test*. Artech House, Inc., Norwood, MA, USA, 1st edition, 2000.

16. Jain Tarush and Agrawal Tanmay. The Haswell microarchitecture - 4th generation processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

17. ARM Limited. Cortex-A57 software optimization guide. 2016.

18. David Kanter. Intel's Haswell CPU microarchitecture. *Real World Technologies*, November 2012. URL: http://www.realworldtech.com/haswell-cpu/.

19. S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

How to approach the peak Flops/sec rate? Comparison of x86 and ARMv8 architectures

V.P. Nikolskiy[1,2], V.V. Stegailov[2]

National Research University "Higher School of Economics"[1],
Joint Institute for High Temperatures of the RAS[2],

In this paper we compare the capabilities of compilers to build a vectorized code for modern ARMv8 and x86_64 processors and discuss an assembly algorithm to achieve maximum performance on both architectures. A highly optimized microbenchmark is presented that reaches the maximum fraction of the peak performance of the ARMv8 processor, while such programs for x86_64 architecture have already been developed. The tests have been performed on the system-on-chip Nvidia Tegra X1 with four ARMv8 Cortex-A57 cores.

Keywords: benchmark, vectorization, peak performance, compiler

References

1. V. Voevodin and Vl V. Voevodin. *Parallel'nye vychislenija [Parallel computing]*. BHV-Peterburg SPb, 2002.

2. David A. Patterson and John L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

3. J. Goodacre and A. N. Sloss. Parallelism and the ARM instruction set architecture. *Computer*, 38(7):42–50, July 2005. doi:10.1109/MC.2005.239.

4. David D Pruitt and Eric A Freudenthal. Preliminary investigation of mobile system features potentially relevant to hpc. In *Proceedings of the 4th International Workshop on Energy Efficient Supercomputing*, E2SC '16, pages 54–60, Piscataway, NJ, USA, 2016. IEEE Press. doi:10.1109/E2SC.2016.13.

5. M. A. Laurenzano, A. Tiwari, A. Cauble-Chantrenne, A. Jundt, W. A. Ward, R. Campbell, and L. Carrington. Characterization and bottleneck analysis of a 64-bit ARMv8 platform. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 36–45, April 2016. doi:10.1109/ISPASS.2016.7482072.

6. Ananta Tiwari, Kristopher Keipert, Adam Jundt, Joshua Peraza, Sarom S. Leang, Michael Laurenzano, Mark S. Gordon, and Laura Carrington. Performance and energy efficiency analysis of 64-bit arm using gamess. In *Proceedings of the 2Nd International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '15, pages 8:1–8:10, New York, NY, USA, 2015. ACM. doi:10.1145/2834899.2834905.

7. Yutaka Ishikawa. System software in Post K supercomputer. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. ACM. URL: http://dl.acm.org/citation.cfm?id=2807591.2897785.

8. V. Nikolskiy and V. Stegailov. Efficiency of ARM processors for classical molecular dynamics. *Numerical methods and programming*, 16(4):578–585, 2015. URL: http://num-meth.srcc.msu.ru/english/zhurnal/tom_2015/v16r454.html.

9. V. Nikolskiy and V Stegailov. Floating-point performance of ARM cores and their efficiency in classical molecular dynamics. *Journal of Physics: Conference Series*, 681(1):012049, 2016. URL: http://stacks.iop.org/1742-6596/681/i=1/a=012049.

10. V. P. Nikolskiy, V. V. Stegailov, and V. S. Vecher. Efficiency of the Tegra K1 and X1 systems-on-chip for classical molecular dynamics. In *2016 International Conference on High Performance Computing Simulation (HPCS)*, pages 682–689, July 2016. doi:10.1109/HPCSim.2016.7568401.

11. Vsevolod Nikolskii, Vyacheslav Vecher, and Vladimir Stegailov. Performance of MD-algorithms on hybrid systems-on-chip Nvidia Tegra K1 & X1. *Communications in Computer and Information Science*, pages 199–211, 2016. doi:10.1007/978-3-319-55669-7_16.

12. Alexander Heinecke, Wolfgang Eckhardt, Martin Horsch, and Hans-Joachim Bungartz. *Supercomputing for Molecular Dynamics Simulations*. SpringerBriefs in Computer Science. Springer International Publishing, 2015. doi:10.1007/978-3-319-17148-7.

13. Rahul Garg. *Exploring the Floating Point Performance of Modern ARM Processors*. AnandTech, 2013. URL: http://www.anandtech.com/show/6971/exploring-the-floating-point-performance-of-modern-arm-processors.

14. Darren Cepulis. ARM antes up for an HPC software stack. *The Next Platform*, March 2017. URL: https://www.nextplatform.com/2017/03/15/arm-antes-hpc-software-stack/

15. Rochit Rajsuman. *System-on-a-Chip: Design and Test*. Artech House, Inc., Norwood, MA, USA, 1st edition, 2000.

16. Jain Tarush and Agrawal Tanmay. The Haswell microarchitecture - 4th generation processor. *International Journal of Computer Science and Information Technologies*, 4(3):477–480, 2013.

17. ARM Limited. Cortex-A57 software optimization guide. 2016.

18. David Kanter. Intel's Haswell CPU microarchitecture. *Real World Technologies*, November 2012. URL: http://www.realworldtech.com/haswell-cpu/.

19. S. Williams, A. Waterman, and D. Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.