DOI 10.37882/2223-2966.2023.12.20

ТЕСТИРОВАНИЕ СКОРОСТИ ВЫПОЛНЕНИЯ АРИФМЕТИЧЕСКИХ ОПЕРАЦИЙ С ЦЕЛЫМИ И ДЕЙСТВИТЕЛЬНЫМИ ЧИСЛАМИ ПРИ ВЫПОЛНЕНИИ ВЫЧИСЛЕНИЙ В КОНТЕЙНЕРАХ DOCKER И НЕПОСРЕДСТВЕННО НА ХОСТЕ

PERFORMANCE EVALUATION OF INTEGER AND FLOATING-POINT ARITHMETICAL OPERATIONS CARRIED OUT USING DOCKER CONTAINERS AND HOST MACHINE

I. Lapin
A. Sergeev
A. Khakhina

Summary. This article provides the description and testing results of the performance of arithmetical operations with integer and floating-point numbers carried out using Docker containers with Linux operating system and the host machine with MacOS. Benchmark results and the findings provided in this article might be useful while deciding to use the Docker virtual machines to maintain the simultaneous calculations while considering the possible overheads and increased time consumption.

Keywords: computer technology, containerization, Docker, operating system, computer, virtualization, testing, integer calculations, floating-point calculations, benchmark.

Лапин Игорь Александрович
ФГАОУ ВО Санкт-Петербургский политехнический университет Петра Великого
igarjar.co@gmail.com
Сергеев Анатолий Васильевич
К.т.н., доцент, ФГАОУ ВО Санкт-Петербургский политехнический университет Петра Великого
sergeev_av@spbstu.ru
Хахина Анна Михайловна
Д.т.н, профессор, ФГАОУ ВО Санкт-Петербургский политехнический университет Петра Великого
anna-hahina@mail.ru

Аннотация. ВВ данной статье приводится описание и результаты тестирования производительности арифметических операций с целыми и действительными числами при выполнении вычислений в контейнерах Docker с операционной системой Linux, а также непосредственно на хосте с операционной системой MacOS. Результаты тестирования и приведенные выводы могут быть полезны при принятии решения о возможности использования виртуальных машин Docker для обеспечения одновременных вычислений с учетом появляющихся накладных расходов и увеличения времени вычислений.

Ключевые слова: компьютерные технологии, технологии контейнеризации, Docker, операционная система, компьютер, виртуализация, тестирование, целочисленные вычисления, вычисления чисел с плавающей запятой, оценка производительности.

В настоящее время практика использования Docker-контейнеров стала уже необычайно широкой и популярной. Такие контейнеры используются при разработке, тестировании, развертывании приложений, обеспечении рабочей инфраструктуры, реализации SaaS (Software as a Service) [1] и прочее.

В связи с этим актуальным и интересным является вопрос производительности подобных контейнеров — насколько сильно отличаются показатели производительности при работе непосредственно на хосте с прямым доступом к аппаратным ресурсам, и внутри Docker-контейнера. Один из самых простых способов сравнения — оценка времени выполнения арифметических вычислений ресурсами процессора в контейнере и непосредственно на хосте.

Таким образом, цель данной работы заключается в том, чтобы оценить различия в скорости выполнения арифметических вычислений с целыми и действительными числами непосредственно на хосте и в Docker-контейнере.

Данную задачу можно решать средствами различных языков; в данном случае мы воспользуемся средствами языка C, чтобы обеспечить высокий уровень точности вычислений и наибольшего приближения к изначальным стандартам типов данных, без использования усложненных структур (к примеру, Python использует расширенные варианты C-типов, что также может сказываться на скорости вычислений) и т.п.

Задуманный эксперимент призван оценить «качество» использования вычислительных ресурсов системы хоста при вычислениях непосредственно на хосте, а также при вычислениях в Docker-контейнерах.

Для создания программы на C потребуется воспользоваться компилятором, к примеру — GCC. Выберем та-

кой Docker-image из стандартного репозитория Docker-образов, в котором сразу установлен компилятор — gcc [2]. Этот дистрибутив опирается на Debian.

Так как исследования будут проводиться на компьютере под управлением MacOS Big Sur, то, чтобы обеспечить наибольшую степень схожести между виртуальным и физическим окружениями, Docker-контейнер идеально было бы создать с использованием дистрибутива MacOS. Однако, единственная существующая реализация OSX для Docker-контейнеров — sickcodes/docker-osx [3], которая требует хост, совместимый с виртуальными машинами KVM, к сожалению, не представленными для MacOS. Поэтому воспользуемся контейнером с дистрибутивом Linux (изначальный Unix подход у систем общий).

Итак, перейдем к разработке тестовой программы. Нам необходимо оценить скорость выполнения арифметических вычислений, для этого будем последовательно выполнять различные арифметические операции: сложение, вычитание, умножение и деление. В качестве операндов будут выступать случайно сгенерированные для каждого теста значения, так мы избежим дополнительной оптимизации со стороны компилятора, а также постоянных кэш-попаданий.

Для оценки времени нам потребуется зафиксировать момент времени непосредственно перед выполнением вычислений, а затем сразу после завершения вычислений, и в результате найти временной промежуток между этими моментами. Также поскольку один «прогон» набора арифметических операций над несколькими числами будет выполнен за слишком незначительный промежуток времени, который не окажется показательным, а также с целью повышения стабильности оценки времени вычислений на более длительном временном интервале, вычисления будут выполняться в циклах по 100 миллионов итераций. Основа для кода программы взята со stackoverflow [4] и модифицирована под текущую задачу. Здесь для генерации случайных чисел используется формула (*rand*( ) % 256)/16 + 1, в которой в скобках генерируются случайные числа от 0 до 255, а затем диапазон сужается до [1;16]. Можно было бы сразу получать числа от 0 до 16 в первых скобках, однако используя формулу в таком виде мы получаем более равномерное распределение.

Для более точной оценки времени производительности будем запускать тестовую программу несколько раз подряд, а затем вычислим средние показатели. Так как нам потребуется в дальнейшем производить такие запуски одновременно в нескольких контейнерах необходимо автоматизировать данный процесс. Сделать это можно, например, при помощи bash-скрипта. Будем выполнять программу 10 раз подряд, записывать получаемые промежуточные результаты как кумулятивную сумму, а затем вычислять средние значения и выводить их в stdout для фиксации и занесении в сравнительные таблицы.

Теперь можно перейти непосредственно к испытаниям. Все испытания, освещаемые в рамках данной работы, проводились на одном и том же физическом оборудовании при максимально возможном соблюдении равносильной нагрузки во время проведения всех тестов, не связанной напрямую с проводимыми тестами. Испытания проводились на персональном компьютере, поэтому, потенциально, результаты испытаний могут быть не столь показательны для оценки возможной относительной производительности в корпоративной среде на специализированном оборудовании. Технические характеристики системы, на которой проводились все испытания, представлены в таблице 1. Стоит отметить, что приводятся те характеристики, которые актуальны при решении конкретных вычислительных задач, определенных в рамках данной работы (к примеру, видеоадаптер не учитывается, так как видеопамять не будет задействована).

Таблица 1.

Технические характеристики системы

Аппаратный элемент	Характеристики
CPU	Intel Quad Core i5 1038NG7, 4 cores, 8 threads, 2.0GHz stock, 3.8GHz Turbo Boost
RAM	16Gb LPDDR4X SDRAM, 3733MHz
Storage	NVME SSD, 512Gb
OS	MacOS BigSur

Стоит отметить также, что для компиляции тестовой программы мы использовали компилятор gcc с параметрами по умолчанию. При таких настройках компиляция проводится в режиме совместимости с обобщенным семейством процессоров x86-64, поэтому используется стандартный набор инструкций. Только для чисел с плавающей запятой используются немного более продвинутые ассемблерные инструкции семейства Streaming SMID Extensions (SSE) [5]. С такими параметрами не используются инструкции семейства Advanced Vector Extensions (AVX) [6] (включая AVX2 и AVX-512F). Однако, даже если бы мы явно компилировали с учетом поддержки этих инструкций, то векторные инструкции в нашем приложении были бы применены лишь для вычислений чисел с плавающей запятой, и не в полном объеме (по умолчанию используется 128-битовый регистр). Чтобы компилятор использовал инструкции семейства AVX необходимо изначально писать код на C с использованием специальных типов данных и функций [7]. Однако, набор инструкций может меняться в зависимости от оборудования конкретной системы, ведь это зависит от поколения процессора (его архитектуры), поэтому,

чтобы получить наиболее обобщенный результат тестирования, имеет смысл использовать стандартный набор инструкций.

В целом же разница между вычислениями для целых чисел и для чисел с плавающей запятой заключается в том, что для них используются различные вычислительные модули: Arithmetic Logic Unit (ALU) [8] для целочисленных операций и Floating-point Unit (FPU) [9] для операций с числами с плавающей запятой. Стоит отметить, что для вычисления чисел с плавающей запятой потенциально можно было бы использовать и ALU, однако FPU обладает большей мощностью, что позволяет существенно увеличить производительность таких операций.

В качестве некоторой опорной линии возьмем результаты, которые получим при вычислениях непосредственно на хосте, они приведены в таблице 2.

Таблица 2.

Результаты тестирования производительности при запуске на хосте

	Integer calculations		Floating point calculations	
	sec	%	sec	%
Host	5.187	100 %	6.586	100 %

Итак, теперь, имея некоторый baseline, появляется возможность сравнить с ним получаемые результаты в контейнерах. Стоит отметить, что вычисления выполняются последовательно, поэтому нагрузка в нашем случае всегда будет идти на один поток. Таким образом Docker-контейнер всегда будет занимать не больше одного логического ядра (относительно нагрузки со стороны выполняющегося в нем теста). Таким образом мы можем выделить виртуальной машине Docker не все логические ядра, что есть на физической машине, это сохранит не затронутыми некоторые ресурсы хоста для стабильной поддержки работы Docker и системы хоста, в то же время это никак не повлияет на результаты эксперимента, так как что на хосте, что в контейнерах, вся нагрузка будет идти на один поток. Выставим для Docker 4 логических ядра из 8. Сперва протестируем скорость вычислений при одном запущенном контейнере. Результаты теста в сравнении с опорной линией приведены в таблице 3.

Таблица 3.

Результаты тестирования производительности при одном запущенном контейнере

	Integer calculations		Floating point calculations	
	sec	%	sec	%
Host	5.187	100 %	6.586	100 %
Container	5.606	108 %	7.175	110 %
Relative changes	0.419	8 %	0.589	9 %

Можно наблюдать, что время вычислений, пусть и не сильно, но выросло. Это вполне ожидаемый результат, так как для выполнения вычислений в контейнере необходимо сперва направить инструкции ядру гостевой ОС, а затем Docker транслирует их ядру ОС хоста. Очевидно, что на эти операции тратится некоторые время, в результате чего возникает некоторый input-lag, увеличивающий время вычислений, что мы и наблюдаем.

Теперь усложним задачу, запустив одновременно несколько контейнеров. Чтобы результаты теста были наиболее достоверными, воспользуемся для этого механизмом репликации Docker, с помощью которого будет возможно запустить все контейнеры практически в один момент времени, за счет чего все тесты будут выполняться одновременно. Для такой реализации укажем в docker-compose.yml режим работы «replicated» и количество «реплик» контейнера. Будем одновременно запускать сперва 2, затем 3, 5 и 10 контейнеров, чтобы проследить динамику изменения производительности вычислений. Результаты тестов представлены соответственно в таблицах 4, 5, 6 и 7. Возможны небольшие погрешности по времени старта контейнеров даже при запуске с использованием механизма репликации, поэтому во всех экспериментах запуски будут производиться несколько раз, а затем будут посчитаны средние величины времени работы для всех тестов в контейнерах по всем запускам. Для этого будем использовать bash-скрипт, описанный ранее. Это позволит отчасти нивелировать погрешность, однако, она все еще может наблюдаться, так как даже при batch-запуске контейнеров какие-то из них все равно запустятся на доли секунд раньше, чем другие, что, очевидно, может сказываться на времени выполнения, так как другие контейнеры еще не занимают часть ресурсов. На рисунке 1 представлен график итоговых полученных кривых изменения производительности (скорости вычислений) в зависимости от числа одновременно запущенных контейнеров.

Таблица 4.

Результаты тестирования производительности при двух запущенных контейнерах

	Integer calculations		Floating point calculations	
	sec	%	sec	%
Host	5.187	100 %	6.586	100 %
Containers average	5.757	111 %	7.209	109 %
Relative changes	0.570	11 %	0.623	9 %

По данным тестирования можно сделать вывод о том, что один одновременно работающий контейнер не сильно увеличивает время выполнения вычислений относительно выполнения на хосте; для целых чисел накладки составляют 8%, а для чисел с плавающей запятой — 9% (таблица 2). Подобные изменения вполне возможно списать на задержку из-за преобразования команд внутри контейнера к реальному оборудованию.

Таблица 5.

Результаты тестирования производительности при трех запущенных контейнерах

	Integer calculations		Floating point calculations	
	sec	%	sec	%
Host	5.187	100 %	6.586	100 %
Containers average	6.226	120 %	7.624	116 %
Relative changes	1.039	20 %	1.038	16 %

Таблица 6.

Результаты тестирования производительности при пяти запущенных контейнерах

	Integer calculations		Floating point calculations	
	sec	%	sec	%
Host	5.187	100 %	6.586	100 %
Containers average	8.213	158 %	9.422	143 %
Relative changes	3.026	58 %	2.836	43 %

Ранее уже упоминалось, что для BM Docker было выделено 4 ядра, таким образом мы смогли наблюдать, что при выполнении однопоточных операций внутри контейнера, при одновременной работе нескольких контейнеров, если их количество не превышает количество выделенных ядер (логических) BM Docker, то, согласно полученному графику на рисунке 1, снижение производительности наблюдается в пределах 10 % (относительно запусков в контейнерах) (таблицы 3–5). Также можно увидеть, что, действительно, график при приращении количества контейнеров, не превышающем количество логических ядер, остается пологим и медленно растет.

Таблица 7.

Результаты тестирования производительности при десяти запущенных контейнерах

	Integer calculations		Floating point calculations	
	sec	%	sce	%
Host	5.187	100 %	6.586	100 %
Containers average	14.973	289 %	17.859	271 %
Relative changes	9.786	189 %	11.273	171 %

При анализе вычислений с одновременным использованием большего количество контейнеров, чем число выделенных ядер для BM Docker, мы ожидаемо наблюдаем увеличение времени вычислений для каждого контейнера, так как теперь они вынуждены разделять ресурсы между собой. При увеличении количества

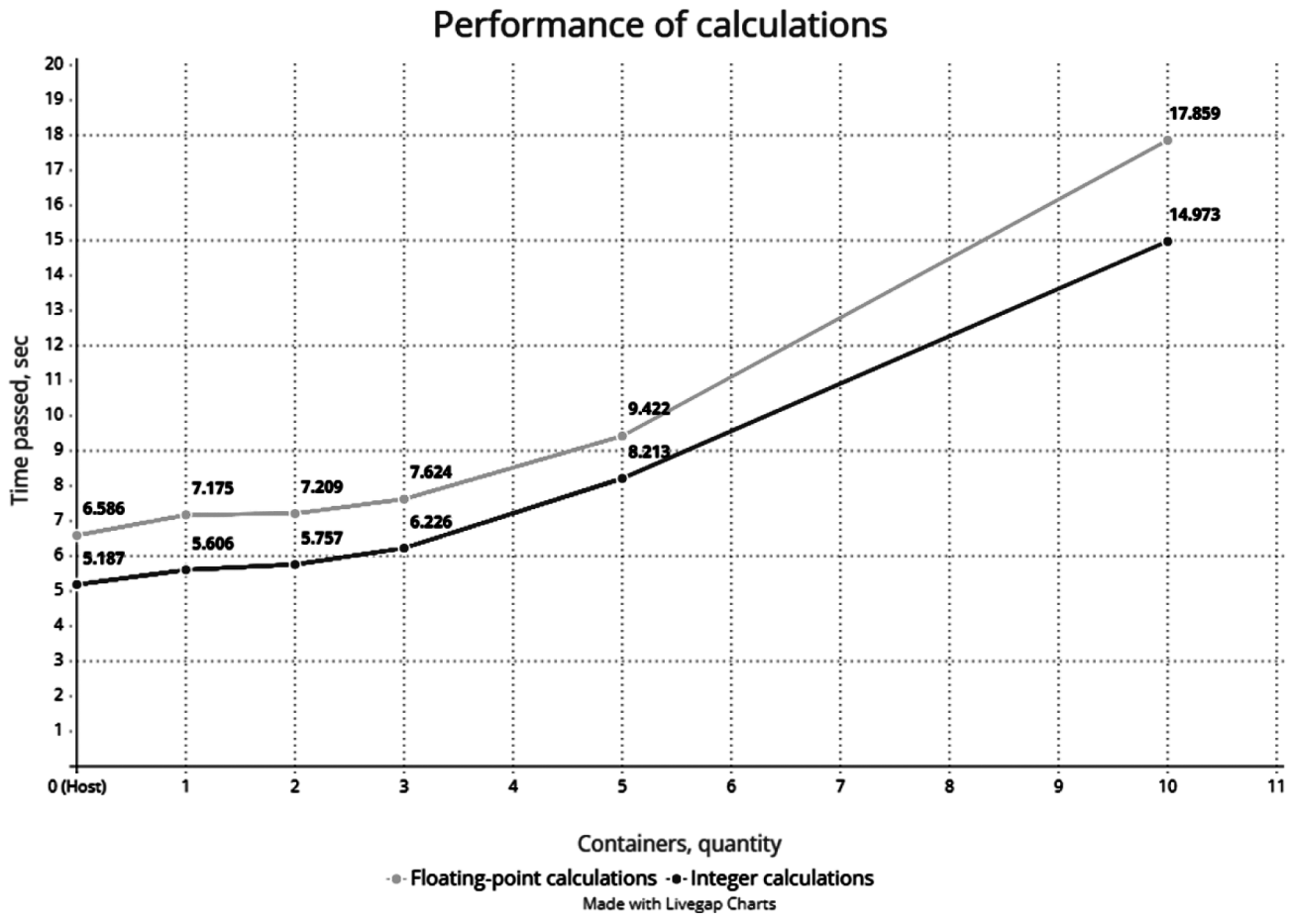

Рис. 1. Итоговые кривые производительности

контейнеров до 5 (таблица 6), а затем до 10 (таблица 7) загруженность ВМ docker составляла 400 % (100 % — одно ядро ВМ); и мы можем отметить, что так как вычисления были начаты одновременно (с небольшой погрешностью на накладки при пакетном запуске), и завершены они также были практически одновременно, это значит, что Docker распределял ресурсы ВМ равномерно между всеми контейнерами. Таким образом, мы можем вычислить, что в среднем каждому контейнеру было выделено $\frac{400\%}{10} = 40\%$ ресурсов. Можно было бы предположить, что в этом случае производительность, относительно одного контейнера, должна была бы упасть примерно в 2.5 раза, однако это цифра выходит больше, — в действительности мы получаем снижение производительности в 2.6–2.8 раза (соответственно для чисел с плавающей запятой и для целых чисел) относительно вычислений в одном контейнере (использованы данные из таблиц 3 и 7. В итоге можем сделать вывод о том, что разница в 10–20 % объясняется растущими накладными расходами на поддержание одновременно большего количества контейнеров. С учетом выводов касательно одновременной работы количества контейнеров, которое не превышает и превышает выделенное ВМ Docker число ядер, следует отметить, что для получения оптимального опыта использования технологии контейнеризации при проведении однопоточных ресурсоемких вычислений не следует использовать в одновременной работе больше контейнеров, чем выделено ядер.

ЛИТЕРАТУРА

1. Oracle — What is a SaaS: website. — URL: https://www.oracle.com/nl/applications/what-is-saas/ (accessed: 04.10.2023)
2. Dockerhub — gcc : website. — URL: https://hub.docker.com/_/gcc (accessed: 19.10.2023)
3. Dockerhub — docker-osx: website. — URL: https://hub.docker.com/r/sickcodes/docker-osx (accessed: 04.10.2023)
4. Stackoverflow — Floating point vs integer calculations on modern hardware: website — URL: https://stackoverflow.com/a/2550851 (accessed: 19.10.2023)
5. Wikipedia — Streaming SIMD Extensions: website. — URL: https://ru.wikipedia.org/wiki/SSE (accessed: 01.11.2023)
6. Wikipedia — Advanced Vector Extensions: website. — URL: https://en.wikipedia.org/wiki/Advanced_Vector_Extensions (accessed: 01.11.2023)
7. Code Project — Crunching Numbers with AVX and AVX2: website. — URL: https://www.codeproject.com/Articles/874396/Crunching-Numbers-with-AVX-and-AVX (accessed: 01.11.2023)
8. Wikipedia — Arithmetic Logic Unit: website. — URL: https://en.wikipedia.org/wiki/Arithmetic_logic_unit (accessed: 25.10.2023)
9. Wikipedia — Floating-point Unit: website. — URL: https://en.wikipedia.org/wiki/Floating-point_unit (accessed: 25.10.2023)