Group 35 - SpecialVFX@Cloud

Lucas Diogo Ferreira Pinto - 110813
Miguel Agostinho da Silva Rocha - 110916
Kenneth Brattli - 1108685

ACM Reference Format:
Lucas Diogo Ferreira Pinto - 110813, Miguel Agostinho da Silva Rocha - 110916, and Kenneth Brattli - 1108685. 2018. Group 35 - SpecialVFX@Cloud. In . ACM, New York, NY, USA, 4 pages. https://doi.org/XXXXXXX.XXXXXXX

1 ARCHITECTURE - MAIN FLOW

The execution starts with the Load Balancer, which is deployed in a single VM, launching, as well, in a separate thread, the AutoScaler.

The data flow starts when a user sends a request to the Load Balancer. The request is processed and subsequently sent to a worker. The worker will either be a VM, or a Lambda function.

The lambdas, as stated in the project description, never instrument code. While VMs can instrument code, some of them are configured to not, for efficiency reasons. The code is instrumented by using a Javassist tool, which modifies the bytecode of the application at compile time, after the VM (and, therefore, the Java program) starts, to insert hooks. These hooks enable the collection of performance metrics.

The workers receive and process the requests, and eventually returning the results to the Load Balancer, which then forwards it to the client.

When a VM Worker starts, it deploys a thread, that, between time to time, flashes all collected metrics (a batch), to the DynamoDB, for efficiency purposes. This avoids the constant access to the DB. Whenever a request is processed by an instrumented VM worker, the collected metrics for that request are cached (which will eventually be sent to the DB, in a batch). Similarly, to enhance efficiency, the Load Balancer caches metrics retrieved from the database, minimizing the need for constant database queries.

2 INSTRUMENTATION METRICS

Currently, only the number of instructions and basic blocks are being collected. One could say that the number of instructions alone would be sufficient to measure the request weight. Yet, it is possible that two different requests produce the same number of instructions, but differ in the number of basic blocks. The request with fewer basic blocks should be considered a lighter request, since it requires fewer optimizations by the JVM machine, duo to the existence of fewer possible execution branches.

For the Image Processing operations (Blur and Enhance), we have found that the number of total executed instructions and basic blocks are a function of the image resolution ($width \times height$). For Raytracer requests the analysis is more complex, and will be addressed in 4.2

3 DATA STRUCTURES

The VM worker caches the request's metrics in a shared (static) data structure, which is also used by the Javassist tool (running in the VM worker) to store the metrics, which groups metrics by thread.

Grouping metrics by thread is important because a thread pool is used by the VM worker web server to process the requests, meaning, threads are re-utilized to fulfil requests. Therefore, each time a request is fulfilled, the metrics of that thread are stored apart from the thread (without the information of the thread), to later be flashed to the DB, and the thread metrics are cleaned, allowing to store metrics of a new request, that is handled by that thread.

Both the request parameters and execution metrics are stored in the DynamoDB. Initially, these were also stored in a file, in order to facilitate the analysis of the algorithms. Yet, this was disabled duo to efficiency purposes. This allows for their later retrieval to estimate the cost of each request.

For both the image processing and Raytracer metrics, the request parameters are stored along with the execution metrics. These include the height and width for image processing, scene width and height, window width and height, and the window row and column offset for ray tracing. When storing these metrics of the DynamoDB, a UUID was used as the primary key, which is random generated for each DB entry. Ideally, this should not be necessary, as the collected metrics should remain consistent for the same parameters, which should serve as the primary key. However, due to a bug encountered when attempting to use multiple parameters as the primary key, we opted to use UUIDs to avoid further delays.

4 REQUEST COST ESTIMATION

Important note: as said in the Instrumentation Metrics section, the number of instructions, as well as basic blocks, should both be used to determine the cost of the request. That is why these two values are collected. Yet, do to time constraints, we didn't have enough time to test the system using these two values. Therefore, the cost estimation for all operations uses only the number of instructions. This should be addressed as future work.

If a specific request has already been instrumented in the past, and the metrics are available in the DynamoDB, the stored number of instructions can be used as the cost of the new request. Otherwise, the request cost needs to be estimated. For this estimation, metrics from similar requests are utilized.

First, the Load Balancer checks if enough time has passed since the last metrics retrieval. If so, it fetches metrics from DynamoDB. This approach helps to avoid constantly fetching new records from DynamoDB whenever a new request cost estimate is needed. Then,

Unpublished working draft. Not for distribution.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Conference'17, July 2017, Washington, DC, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/XXXXXXX.XXXXXXX

Unpublished working draft. Not for distribution.

height	width	number_of_basic_blocks	number_of_instructions
637	960	16116381	553925233
686	960	16610907	572325984
407	612	10287399	353533842
639	960	16136565	554676277
493	740	12445533	427767505

Table 1: Offline instrumentation for the Blur image operation

height	width	number_of_basic_blocks	number_of_instructions
637	960	1581196	42807779
407	612	984700	26115567
493	740	1205549	32293171
720	960	1666852	45039649
768	512	1254052	32803457
678	960	1623508	43910269

Table 2: Offline instrumentation for the Enhance image operation

the Load Balancer uses cached metrics to estimate the request cost. If there are no relevant metrics in the cache, the cost is estimated based on offline metrics.

Due to time constraints, we did not have time to study what is considered a similar request. Therefore, for the Image Proc operations, since the height and width are the only request parameters, requests with the following parameters were used to estimate the cost:

- minHeight = imageHeight - 300;
- maxHeight = imageHeight + 300;
- minWith = imageWidth - 300;
- maxWith = imageWidth + 300.

The same approach was applied to the Raytracer's 6 numbered parameters: Scene height and width, window height and width, and Window row and column offset.

4.1 Image Proc request cost estimate

For a given Image Proc request (request to estimate cost), and for a set of Image Proc previously stored metrics (height, width and number of instructions), the cost of that request is estimated by first computing the number of pixels for each stored request (height x width). The number of instructions per pixel is then calculated and multiplied by the number of pixels of the new request. This results in an estimate of the number of instructions needed for the new request.

If there are no record metrics in the DB, we use an average of number of instructions per pixel, which was computed offline, before deploying the system for production. For the Image proc operations (blur and enhance), we obtained these values by computing the average number of instructions per pixel using offline instrumented metrics. Specifically, we calculated 1054.19 instructions per pixel for the blur operation and 79.90 instructions per pixel for the enhance operation. These values were obtained by instrumenting in an AWS VM, and using some of the images available in the resources folder provided in the base project structure. The detailed metrics for these operations are available in tables 1 and 2, for the blur and enhance operations, respectively.

4.2 Raytracer request cost estimate

For the Raytracer case, since there are multiple parameters (unlike the image proc that only have two, leading to one, which is the number of pixels), it's much harder to compute an estimate cost. Therefore, we conducted an offline test, where we obtained the results of the importance of each parameter for the total number of instructions. An $2^(k-p)$ Fractional Factorial Design experiment

Figure 1: The importance of each parameter of the Raytracer operation, for the cost

Figure 2: The importance of each parameter of the Raytracer scene, for the cost

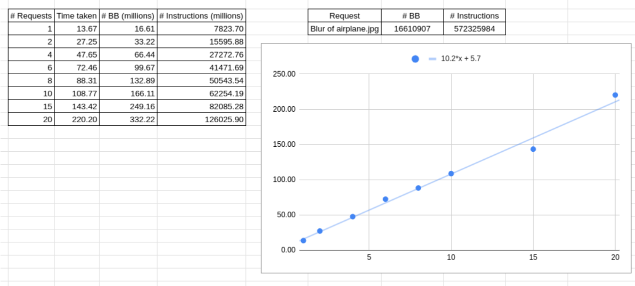
was conducted, with the confounded variables being srows, wrows The results are shown in table 1

With the experimentally obtained parameter costs in 1, the objective is to reach the number of instructions per parameter unit. For that, we multiply the total number of instructions, of a request, by the cost/weight, and through an average we get our solution. This superficial analysis only took into account the top level parameters. Most were numbers, but scene is composed of multiple parameters in itself.

For that, another similar analysis, 2, was performed on the scene's parameters. A more in-depth analysis could be performed, as this one only accounts for the amount of each parameter, and not the inner parameters of each one. Even more, from the results, we only care for lighting as it contributes with almost 95% of the scene's instructions count. We can neglect the other scene parameters because their contribution is residual.

If one wants to estimate the cost of a request, but there are no similar-stored metrics, in order to do the computation above, an average number of requests (cost) is used, which was computed offline, issuing different Raytracer requests, and computing the average of number of instructions (cost). This is not the best approach, since the Raytracer operations have various parameters, being, most of them, numeric values. Yet, note that this is only a

Unpublished working draft. Not for distribution.



# Requests	Time taken	# BB (millions)	# Instructions (millions)
1	13.67	16.61	7823.70
2	27.25	33.22	15595.88
4	47.65	66.44	27272.76
6	72.46	99.67	41471.69
8	88.31	132.89	50543.54
10	108.77	166.11	62254.19
15	143.42	249.16	82085.28
20	220.20	332.22	126025.90

Request	# BB	# Instructions
Blur of airplane.jpg	16610907	572325984

Figure 3: Values obtained in offline tests, performed in an AWS VM instance, in order to study how the number of parallel requests affects the execution time

problem when there are no similar metrics stored, which should become less of a problem with time. The tests are displayed in table ??. The average number of instructions is 361 855 796.

4.3 Finding the maximum AWS instance Workload

In order to estimate the maximum load each instance can handle, tests were performed to determine how long it would take to execute the requested work. The test was performed by issuing parallel requests, and recording the total executed basic blocks, number of instructions, and average of time taken. These tests were conducted in an AWS t2.micro VM, since it's the target environment of the project. The results are shown in ??. We opt to use a workload equivalent of 4 requests, which results in a waiting time of less than a minute, making the total workload per VM worker to be 27.272.000.000 instructions.

5 TASK SCHEDULING ALGORITHM (LOAD BALANCING)

The Load Balancer is deployed as a Web server that handles requests using a thread pool. When a new request arrives, the LB searches for the less loaded VM instance. This is done by iterating across all registered (created) VM instances, skipping the ones that are marked for termination (see the Auto Scaler section), and computing the total cost of that instance, which is done by computing the cost of each individual request assigned to that instance (requests that are or going to run on that instance). The less loaded instance is checked to be valid, by checking if the already accumulated instance cost, plus the one of the new request to execute, does not exceed the total instance workload threshold. If it's valid, the request is sent to that instance. If one VM instance is valid and still being deployed, but not yet running, the thread will wait, before sending the request to the instance. If there isn't any valid instances, the Load Balancer launches a new thread that starts a new VM worker, or a lambda function. If the thread decided to create a VM instance (instead of launching a Lambda function), while that thread is waiting for the new instance to be ready, it's possible that a new instance, previously considered invalid, or one that was already starting, becomes available. Therefore, the original thread continues in a loop, from time to time, checking if there is a new valid instance to send the request, and if so, it sends the request to that instance.

When the new VM instance is finally running, the thread that created it, checks if the request was already fulfilled, and, if not, the request is sent to the newly created VM instance or lambda. Of course, it's possible that a new VM instance was launched to handle the new request, and, after it's running, a previous invalid instance, is now able to handle the request, making the newly created VM instance, useless.

Note that it's possible that many requests arrive almost at the same time, and will be handled in parallel. At first sight, this may appear to be a problem when there are no available instances to handle some of the requests, because all the threads handling each request, see at the same time, that there are no available instances to handle the request), and that this would cause multiple VM instances to be launched, (one per request), which is not desirable, since one Vm could satisfy various requests. Therefore, there is a locking mechanism that prevents multiple requests to launch a VM instance in parallel. Instead, only one thread, the first to acquire the lock, will check if there are no available running instances that could satisfy the request, and, if not, the thread will create launch a VM (if not a lambda), and release the lock without waiting for the instance to start. Only then, the next thread will acquire the lock, and be able to see this newly created instance, if it was indeed created, and assess if it is able to run the thread request.

5.1 Send to Lambda or to a VM instance?

We decided that only the lighter requests would go to a Lambda instance, and only if there are no available instances that could execute the request. Lambdas are more expensive to execute, but they execute faster, when compared with having to wait for a new VM instance to be ready. Therefore, if the request is lighter, it's worth it to send it to a Lambda, since it will execute quicker, spending less money than a heavier request. Also, creating a new VM, in order to execute a lighter request, would result in more time spent for the instance deploy, than the actual request execution, in opposition to a heavier request. A light request is the one whose estimated cost is less than 30% of the average cost of that operation.

6 AUTO-SCALING ALGORITHM

The Auto Scaler runs in an infinite loop, sleeping and waking from time to time. When it wakes, it iterates across all running instances and, for the ones marked for termination, and that currently have no workload (processing 0 requests), terminate that instance. Note that the VM could have a CPU usage of 100%, and still not executing useful work (processing requests).

Then, it iterates across all instances again, ignoring the ones that are marked for termination, or that are still being deployed (not yet running), and marks the ones that are suitable for termination. If an instance has a current workload (computed from the requests that are being executed, at that moment) of less than 30% of the maximum instance workload, and if that instance is not the only one available, (excluding the ones marked for termination), the instance is marked for termination, otherwise, if it's indeed the last one available, it's only marked for termination if the current CPU usage is less than 50%. It's important to have at least one instance running, even with low workload, any new request could arrive,

Unpublished working draft. Not for distribution.

Table 3: Results obtained by issuing some Raytracer requests in an AWS VM, in order to estimate the average number of requests, to be used as an offline heuristic.

#_lights	#_instructions	scene_height	scene_width	window_col_offset	window_height	window_row_offset	window_width
3	779	200	300	0	400	0	300
3	471859042	20	100	10	600	20	200
3	651139215	200	400	10	600	0	200
3	779	200	400	0	400	0	300
3	752060914	200	400	0	600	0	300
3	646221254	200	400	10	600	20	200
3	795	400	300	0	400	0	300
3	971480914	200	400	0	600	0	200
3	638278465	200	100	10	600	20	200

and it wouldn't make sense to have an instance star, execute the request, and die right afterwards.

6.1 Instance Data

For simplicity purposes, the Load Balancer does not query the AWS, in order to know the already available VM Worker instances, during startup. Instead, it is assumed that the Load Balancer is the first to be created, there are no created AWS VM Workers, and that it never fails, as the project description does not mention Load Balancer (and Auto Scaler) failures. Therefore, the Load Balancer starts with an empty set of Worker instances, and populates the set as needed. This set of available instances is shared with the Auto Scaler, since they both use this data structure.

An instance is described by:

- it's ID;
- a set of requests;
- a boolean that specifies if the instance will terminate;
- IP and Port.

Upon creation, these fields are set, except for the ID and IP which are both null. Even without setting up the IP, which is available only when the instance is in the running state, the Load Balancer can already see that this instance is going to run, and the amount of requests the instance has already scheduled, and make decisions even before the instance is in the running state. When the Load Balancer decides to send a request to that instance, it appends the request to the set of requests of the instance, so the other threads, handling other requests, can use the instance already scheduled set of requests to estimate the current instance workload.

7 FAULT-TOLERANCE

Each time a request is sent over to a VM or lambda function, to be executed, in case there is a problem and the response can not be obtained, the request is re-tried again. Also, to avoid any future problems with that instance, where the request failed, the instance is terminated.

Unpublished working draft. Not for distribution.