Architecture-Specific Performance Optimization of Compute-Intensive FaaS Functions

Mohak Chadha*, Anshul Jindal*, Michael Gerndt*
*Chair of Computer Architecture and Parallel Systems, Technische Universität München
Garching (near Munich), Germany
Email: mohak.chadha@tum.de, jindal@in.tum.de, gerndt@in.tum.de

2021 IEEE 14th International Conference on Cloud Computing (CLOUD) | 978-1-6654-0060-2/21/$31.00 ©2021 IEEE | DOI: 10.1109/CLOUD53861.2021.00062

***Abstract*—FaaS allows an application to be decomposed into functions that are executed on a FaaS platform. The FaaS platform is responsible for the resource provisioning of the functions. Recently, there is a growing trend towards the execution of compute-intensive FaaS functions that run for several seconds. However, due to the billing policies followed by commercial FaaS offerings, the execution of these functions can incur significantly higher costs. Moreover, due to the abstraction of underlying processor architectures on which the functions are executed, the performance optimization of these functions is challenging. As a result, most FaaS functions use pre-compiled libraries generic to x86-64 leading to performance degradation. In this paper, we examine the underlying processor architectures for Google Cloud Functions (GCF) and determine their prevalence across the 19 available GCF regions. We modify, adapt, and optimize three compute-intensive FaaS workloads written in Python using Numba, a JIT compiler based on LLVM, and present results wrt performance, memory consumption, and costs on GCF. Results from our experiments show that the optimization of FaaS functions can improve performance by 12.8x (geometric mean) and save costs by 73.4% on average for the three functions. Our results show that optimization of the FaaS functions for the specific architecture is very important. We achieved a maximum speedup of 1.79x by tuning the function especially for the instruction set of the underlying processor architecture.**

***Index Terms*—Function-as-a-service (FaaS), serverless computing, performance optimization, cost, heterogeneity, Numba, LLVM**

I. Introduction

Since the introduction of AWS Lambda by Amazon in 2014, serverless computing has grown to support a wide variety of applications such as machine learning [1], map/reduce-style jobs [2], and compute-intensive scientific workloads [3], [4], [5]. Function-as-a-Service (FaaS), a key enabler of serverless computing allows a traditional monolithic application to be decomposed into fine-grained functions that are executed in response to event triggers or HTTP requests on a FaaS platform. The FaaS platform is responsible for the isolation and execution of these functions in dedicated function instances, usually containers.

FaaS platforms follow a process-based model for resource management, i.e., each function instance has a fixed number of cores and quantity of memory associated with it [6]. While today's commercial FaaS platforms such as Lambda, GCF abstract details about the backend infrastructure management away from the user, they still expose the application developers to explicit low-level decisions about the amount of memory to allocate to a respective function. These decisions affect the provisioning characteristics of a FaaS function in two ways. First, the amount of CPU provisioned for the function, i.e., some providers increase the amount of compute available to the function when more memory is assigned [7]. Selecting an appropriate memory configuration is an optimization problem due to the trade-offs between decreasing function execution time with increasing memory configuration and costs. Moreover, assigning more memory than desired can lead to significant resource over-provisioning and reduced malleability [8]. Second, the addition of a per-invocation duration-utilization product fee measured in GB-Second (and GHz-Second with GCF [9]). FaaS is advertised as a pay-per-use model, where the users are billed based on the execution time of the functions measured typically in 100ms (GCF) or 1ms (Lambda) intervals. As a result, for compute-intensive functions that require more than the minimum amount of memory the duration-utilisation component fee can lead to significantly higher costs.

While compute-intensive applications are written in a wide variety of high-level languages such as Java, R, and Julia. In this paper, we focus on Python since it is a widely used high-level programming language for compute-intensive workloads [3]. Furthermore, it is supported by all major commercial FaaS platforms. To facilitate the performance improvement of applications written in Python several approaches exist. These include using an alternative Python interpreter such as PyPy or using a Python to C/C++ transpiler such as Cython. Using a replacement Python interpreter has the disadvantage that it has it's own ecosystem of packages which are significantly limited. Disadvantages of using a transpiler is that it offers limited static analysis, and that the code has to be compiled Ahead-of-Time (AOT). This leads to under-specialized and generic code for a particular CPU's architectural family (such as `x86-64`) or can cause code bloating to cover all possible variants [10]. To this end, we utilize Numba [11], a function-at-a-time Just-in-Time (JIT) compiler for Python based on LLVM [12] for optimizing and improving the performance of compute-intensive FaaS functions. Using Numba has several advantages. First, it gets around the native Python interpreter (CPython) and generates machine code. Second, LLVM compiler optimizations and support for special instruction set extensions based on the underlying processor architecture such as AVX-2/AVX-512. Third, gets around the Global-interpreter Lock (GIL) and supports multiple threading backends (Intel

2159-6190/21/$31.00 ©2021 IEEE
DOI 10.1109/CLOUD53861.2021.00062

Authorized licensed use limited to: University of Washington Libraries. Downloaded on November 07,2023 at 08:00:59 UTC from IEEE Xplore. Restrictions apply.

TBB, OpenMP). Finally, it is easy to use and involves simply decorating the Python function with the Numba decorator (`@njit`) to get good performance.

On invocation of a deployed function, the function instances are launched on the FaaS platform's traditional Infrastructure as a Service (IaaS) virtual machines (VM) (microVMs in Lambda) offerings. However, the provisioning of such VMs is abstracted away from the user. As a result, the user is not aware of the details of the provisioned VMs such as the CPU architecture and the number of virtual CPUs (vCPUs). This makes performance optimization of FaaS workloads challenging.

Identification of the set of architectures dynamically used in current commercial FaaS platforms is important for the performance optimization of FaaS functions. Previous works [6], [7] have reported the presence of Intel based processors ranging from Sandy Bridge-EP to Skylake-SP architectures in the provisioned VM. However, due to the rapid development in FaaS offerings of major cloud providers, and to offer updated insights, we investigate the current CPU processor architectures for GCF.

Our key contributions are:

- We investigate the current CPU architectures present in GCF across the different regions.
- We analyze the impact of heterogeneity in the underlying processor architectures on the performance of a FaaS function.
- We modify, adapt, and optimize three FaaS workloads[1] from FunctionBench [4], and the Python performance benchmark suite (Pyperf) [13] using Numba.
- We deploy the optimized workloads on GCF for the different memory profiles and analyze the impact on performance, costs, and memory consumption.

II. Related Work

FaaS Optimizations. Majority of the previous works [14], [15] have focused on optimizing the cold start problem associated with FaaS. Mohan et al. [14] identify the creation of network namespaces during container startup as the major reason for overhead for concurrent function invocations. Towards this, they propose the usage of Pause Containers (PCs), i.e., a set of pre-created containers with cached networking endpoints, thereby removing network creation from the critical path. Fuerst et al. [15] develop FaasCache, based on OpenWhisk, that implements a set of caching-based keep-alive policies for reducing the overhead due to function cold-starts. In contrast to previous works, we optimize the performance of a representative set of common FaaS workloads and present benefits/tradeoffs in terms of performance, memory consumption, and costs when deployed on a public cloud provider, i.e., GCF.

Understanding the Backend Infrastructure in Commercial FaaS Platforms. The most notable works in this domain have been [6], [7]. Wang et al. [6] performed an in-depth study of resource management and performance isolation with three popular serverless computing providers: AWS Lambda, Azure Functions, and GCF. They show that the provisioned VMs across the different platforms have great heterogeneity wrt the underlying processor architectures and configuration such as number of virtual CPUs. Kelly et al. [7] provide an updated view on the VM toplogy of the major FaaS platforms including IBM Cloud Functions. While these previous works have inspired some of the methodology of the experiments used in this work, there are some key differences. First, we identify the prevalence of different processor architectures in the provisioned VMs across the 19 different available GCF regions. Second, we demonstrate how the underlying VM configuration such as the number of vCPUs can be used for optimizing the performance of functions. Third, we demonstrate the effect of microarchitectural differences in the underlying processor architectures on the performance of FaaS functions.

[1]https://github.com/kky-fury/Optimizing_FaaS_Workloads

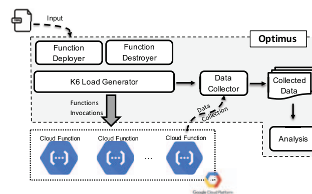


Fig. 1: Architecture of our benchmarking and data acquisition tool *Optimus*.

III. Methodology

To facilitate the deployment, deletion, benchmarking, and metric data acquisition of functions on GCF, we have developed *Optimus*. It's architecture and different components are shown in Figure 1. *Optimus* takes a YAML file as input that specifies the GCF function configuration parameters for the function deployment, the function to be deployed, and configuration parameters for the load generator. Following this, the *Function Deployer* which encapsulates the functionality of the `gcloud function` command-line tool deploys the function according to the specified parameters. To invoke and evaluate the performance of the deployed function, we use `k6`. `k6` is a developer-centric open-source load and performance regression testing tool. As part of each `k6` test, two additional parameters are configured, i.e., Virtual Users (VUs), and test duration which can be specified in the input YAML file. To collect the metric data on completion of a function load test, we implement a monitoring client using the Google Cloud client library. The different monitoring metrics extracted as part of each test are shown in Table I.

The individual FaaS workloads used in this work and the suites to which they belong are shown in Table II. The *Image processing* application uses the Python `Pillow` library to blur a RGB image using the Gaussian Kernel and then converts the blurred image to grayscale. Following this, the Sobel operator is applied to the grayscale image for edge detection. After completion of the function the modified images are written to a block storage. The *Montecarlo* function calculates the area of a disk by assigning multiple random values to two variables to generate multiple results and then averages the results to obtain

Authorized licensed use limited to: University of Washington Libraries. Downloaded on November 07,2023 at 08:00:59 UTC from IEEE Xplore. Restrictions apply.

TABLE I: Collected GCF monitoring metrics. The metric data is sampled every 10 seconds.

Metric	Description
Active instances	The number of active function instances.
Function Invocations	The number of function invocations.
Allocated Memory	Configured function memory
Execution time	The mean execution time of the function
Memory usage	The mean memory usage of the function.

TABLE II: FaaS workloads used and optimized.

Name	Input	Suite
Image processing	JSON with image urls	FunctionBench [4]
Montecarlo	JSON with number of iterations	PyPerf [13]
Kerneldensityestimate (KDE)	JSON with distribution size, kernel bandwidth, and evaluation point	Other

an estimate. The KDE workload uses the gaussian kernel to estimate the density function. The native implementation is written using Numpy. On completion, it returns the calculated density estimate at the evaluation point.

To optimize and maximize the performance of the FaaS workloads using Numba we refactored the native implementations of the workloads to enable automatic optimization. Towards this, we made use of different decorators supported by Numba such as `@stencil` and additional libraries such as Intel Short Vector Math Library (SVML), and Intel TBB. An important aspect of optimizing compute-intensive functions is vectorization of loops to generate Single Instruction Multiple Data (SIMD) instructions. The LLVM backend in Numba offers auto-vectorization of loops as a compiler optimization pass. On successful vectorization, the compiler will generate SIMD instructions depending on underlying processor's supported SIMD instruction set such as Advanced Vector Extensions (AVX)-2, AVX-512. In this paper, we use the Intel TBB library as a threading backend supported by Numba to parallelize the *Montecarlo*, and individual kernels (gaussian blur, and RGB to gray conversion) of the *Image processing* workload. For all workloads, we also added the argument `fastmath=True` to the `@njit` decorator. This relaxes the IEEE 754 compliance for floating point arithmetic to gain additional performance. Furthermore, it permits reassociation of floating point operations which allows vectorization.

IV. Platform Architecture

The GCF service is regional, i.e., the infrastructure on which the function instance is launched varies across the different available regions [16]. To investigate the different underlying processor architectures of the provisioned VMs across the 19 available GCF regions we used the `proc` filesystem on Linux. Table III shows the different attributes we read from the Linux `procfs`. We implemented a function that reads the attributes and collates them into a JSON response. Following this, we deployed the function for the different supported memory profiles at the time of the experiments[2] across all the available regions using the function deployer component in *Optimus*. We fixed the number of VUs and the duration of the test in `k6` to 60 and 1 minute respectively. As a result, multiple function instances were launched simultaneously to handle the requests. We repeated the `k6` load test every two hours and collected

[2]The experiments were performed in Feb-March 2021.

TABLE III: Data collected from the `proc` filesystem of the provisioned VM on GCF.

Attribute	System Information
vCPUs	Number of virtual CPUs configured in the VM.
CPU Model	CPU model present in the VM.
CPU Family	Family of processors to which the CPU belongs.
Total Memory	Total memory configured in the VM.

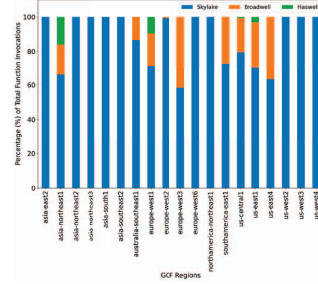


Fig. 2: The different Intel processor architectures across the 19 available GCF regions along with percentage of functions invoked on them.

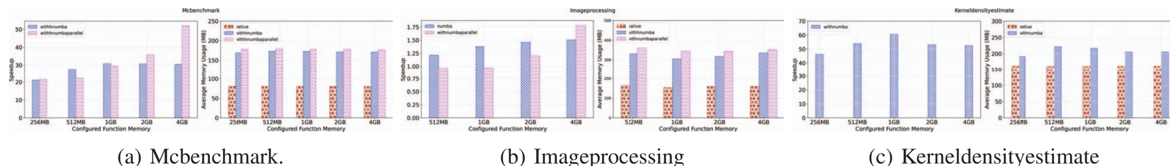
the measurements for a period of two weeks, leading to more than a billion function invocations.

From the collected data, we found that across all regions the VMs provisioned were based on Intel Xeon CPUs. Although Google uses a proprietary hypervisor for running the function instances which hides the model name attribute from the Linux `procfs`, we were able to infer the different processor architectures using the model and family attributes [17]. Particularly, we found three different models from the same family `6`, i.e., `85`–Skylake, `79`–Broadwell, and `63`–Haswell.

In contrast to the results reported by [6], [7], we did not find the architectures `(62,6)`–IvyBridge, `(45,6)`–SandyBridge on any of the provisioned VMs across all GCF regions. We believe since these models were launched in 2013 and 2012 respectively, they have been phased out. Figure 2 shows the prevalence of the different architectures we found across the 19 available GCF regions. For a particular region, we combined the results for all the memory profiles. We found that Intel Skylake was the most prevalent architecture across all regions. Only for the regions `asia-northeast1`, `europe-west1`, `us-central1`, and `us-east1` we found function instances being launched on VMs with all the three processor architectures. We found the greatest heterogeneity in the `asia-northeast1` region. For all regions, we found that irrespective of the configured memory profile the VMs were configured with 2GB of memory and 2 vCPUs. This was also true for a function configured with 4GB of memory. As a sanity check, we wrote a simple function which allocates 3GB of memory when the function is configured with 4GB. This results in a heap allocation error. We believe that this is a bug and have reported it to Google.

While the Intel Skylake processor has several new microarchitectural features, which increase performance, scalability, and efficiency as compared to the Broadwell and Haswell architectures [18], in this paper, we focus only on differences in the SIMD instruction set. The Intel Skylake processor

Authorized licensed use limited to: University of Washington Libraries. Downloaded on November 07,2023 at 08:00:59 UTC from IEEE Xplore. Restrictions apply.



(a) Mcbenchmark. (b) Imageprocessing (c) Kerneldensityestimate

Fig. 3: The obtained speedup and average memory consumption of the three optimized FaaS workloads as compared to their native implementations for the different memory configurations on GCF. All functions are deployed on the `us-west2` region.

supports the AVX-512 SIMD instruction set as compared to AVX-2 in both Broadwell and Haswell architectures. This means that each SIMD unit in Skylake has a width of 512 bits as compared to 256 bits in Broadwell and Haswell which translates to increased FLOPs/cycle and improved performance. On successful autovectorization, the LLVM backend compiler used in Numba will try to generate SIMD instructions based on the highest available instruction set.

V. Experimental Results

A. Experimental Configuration

To compare the optimized and the native FaaS workloads wrt performance, memory consumption, and costs we deploy both versions on the `us-west2` GCF region for all the available memory profiles using *Optimus*. For all workloads, we set the maximum number of function instances to 50 and the timeout to 300 seconds. We chose `us-west2` since it was one of the regions where we observed homogeneous processor architecture, i.e., Skylake in the provisioned VMs (§IV). As configuration parameters to `k6`, we set the maximum number of VUs to 50 and total duration of the load test to five minutes. For all our experiments, we repeated the `k6` test five times every two hours and then averaged the results.

For all the optimized FaaS workloads, we enabled file-based caching of the compiled function machine code generated by Numba. We modified the Numba configuration to save the cached code in `/tmp` filesystem available for GCF. This was done to ensure that function instances provisioned on the same VM have access to the compiled machine code to avoid overhead due to recompilation. This behaviour was first reported by [7], where functions executing on the same VM could read a unique id written to a file in the `tmp` filesystem. From our experiments, we observed that caching improved the speedup by 1.2x on average as compared to the non-cached version. The speedup was not much more significant because Numba jitted functions are stored in memory and retain their state between warm invocations. This means that recompilation of a Numba jitted function (with same function argument types) only occurs with a function cold start. Moreover, for the parallelized FaaS functions, we configured the number of TBB threads to two due to the availability of two vCPUs.

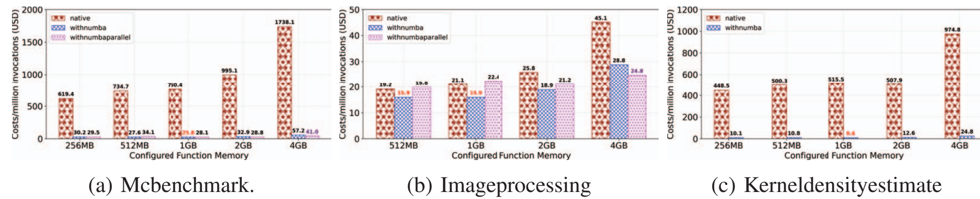
B. Comparing performance and memory consumption

For comparing the performance of the optimized FaaS workloads with their native implementations, we calculate the metric speedup. This is done by dividing the obtained average execution time of the native implementation by the obtained average execution time of the optimized workload for a particular GCF memory configuration. For a particular function and GCF memory configuration, the average execution time is obtained by calculating the weighted average of the number of function invocations and the mean execution time of the function (see Table I). To compare memory consumption, we use the default GCF monitoring metric, i.e., Memory usage and average it across all the available datapoints. The obtained speedup and average memory usage for the different workloads for the different available GCF memory configurations is shown in Figure 3. We report all performance results for double precision floating point operations.

We obtained an average speedup of 28x, 32x for the single-threaded and parallelized versions of the *Mcbenchmark* across the different memory configurations as shown in Figure 3a. The main reason for the significant increase in the performance of the FaaS functions optimized with Numba is the generation and execution of machine code. On the other hand, for the native FaaS function, Python automatically generates bytecode which is executed by the default bytecode interpreter. Although the underlying provisioned VMs are configured with two vCPUs, we do not observe an increase in speedup for the parallel function as compared to the single-threaded function for all memory configurations. This is because GCF uses a process-based model for resource management, where each function has a fixed memory and allocated CPU cycles. Since Intel-TBB follows a *fork-join* model for parallel execution, the generated threads are inherently limited by the resource constraints of the parent process. We observe that the speedup of the parallelized function as compared to the single-threaded version increases with the increase in the allocated CPU clock cycles, i.e., when more memory is configured.

For the *Image Processing* workload, we obtained an average speedup of 1.40x, 1.23x across the different memory configurations for the single-threaded and parallelized versions respectively. The speedup values obtained are comparatively small since the native implementation of the benchmark uses the Python `Pillow` library. As shown in Figure 3b, the single-threaded Numba optimized *Image processing* function performs better than the native implementation due to LLVM compiler optimizations, and vectorization using the highest underlying SIMD instruction set. In contrast, `Pillow` is pre-compiled and generic to `x86-64`. This means that the vector instructions generated will be for the Streaming SIMD Extensions (SSE) instruction set. The parallelized Numba optimized function performs worse than the native implementation for the memory configurations 512MB, 1GB, due to limited CPU clock cycles and parallelization overhead. For

Authorized licensed use limited to: University of Washington Libraries. Downloaded on November 07,2023 at 08:00:59 UTC from IEEE Xplore. Restrictions apply.



(a) Mcbenchmark. (b) Imageprocessing (c) Kerneldensityestimate

Fig. 4: Comparison of cost per million function invocations (in USD) of the three FaaS workloads as compared to their native implementations for the different memory configurations on GCF. The cost values highlighted with red represent the minimum values obtained across the different memory configurations, while the cost values highlighted with purple (if present and different) represent the values wrt the maximum percentage cost savings.

the optimized *Kernel Density Estimate* functions we observe an average speedup of 53x across the different GCF memory configurations respectively. We observe a maximum speedup of 61x for the optimized KDE functions for the memory configuration of 1GB as shown in Figures 3c.

For all benchmarks, we observe that the average memory usage of the Numba optimized functions is higher than their native implementations as shown in Figures 3a, 3b, and 3c. This can be attributed to (i) additional variables required for Numba's internal compilation workflow, (ii) additional module dependencies such as LLVM, `icc_rt`, and (iii) in-memory caching of the generated machine code. The memory required for the Numba parallelized functions is more as compared to the single-threaded functions because of the additional `intel-tbb` library. Note that, due to the presence of coarse grained memory profiles and billing policy adopted by GCF [9], users will be charged based on the configured memory, irrespective of the function memory usage.

C. Comparing costs

Figure 4 shows the cost per million invocations of the optimized FaaS workloads as compared to their native implementations for the different memory profiles on GCF. To compute the invocation cost of a particular function and GCF memory configuration, we follow the rules and pricing values specified by Google [9]. We observe 96.2%, 96.4% average cost savings for the two Numba optimized functions of the *Mcbenchmark*. The minimum cost value of $25.8 is obtained for the single threaded function when configured with 1GB of memory as shown in Figure 4a. The maximum cost savings of 97.64% is obtained with a memory configuration of 4GB for the parallelized function.

We observe 26.1% average cost savings for the single-threaded *Image processing* function across the different memory configurations. The cost values obtained for the parallelized function are higher as compared to the native implementation for the memory configurations 512MB and 1GB respectively. But, they decrease when higher memory is configured as shown in Figure 4b. We observe 97.75% average cost savings for the optimized *KDE* function across the different memory configurations. The minimum cost value and maximum cost savings of $9.6 and 98.1% are obtained for the memory configuration of 1GB as shown in Figure 4c.

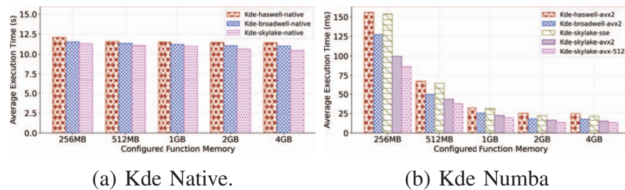
Although the speedup obtained for the different optimized function varies across the different memory configurations, we do not observe a significant difference in costs for the Numba optimized functions across the memory configurations as shown in Figure 4. GCF offers the possibility of unlimited scaling of function instances to meet user demand [19]. To avoid memory over-provisioning and due to the significant speedup obtained with Numba for the lowest possible memory configuration for a particular function, the minimum memory configuration can always be selected. Moreover, we observe that parallelization of functions is only beneficial when configured with a memory of 2*GB* and higher because of constraints on the allocated CPU clock cycles.

D. Effect of heterogeneity in the underlying processor architectures on performance

To analyze the effect of different processor architectures on the performance of a FaaS function, we use the *Kernel Density Estimate* (KDE) workload and deploy it for all supported memory configurations in the `asia-northeast1` region. We chose this region since it had the greatest heterogeneity and prevalence of the three processor architectures. We instrumented the KDE workload to compute the execution time required for calculating the estimate at the evaluation point given as input. The processor architecture is determined similarly as described in §IV. As described in §III, Numba automatically generates SIMD instructions for highest underlying instruction set. However, to emphasize the importance of generating architecture-specific code, we modified the Numba configuration to generate only AVX-2 and SSE instructions on the Skylake processor. Figure 5b shows the average execution time for the different processor architectures and SIMD instruction sets across the different memory configurations for the Numba optimized *KDE* function.

For all processor architectures the average execution time decreases with increasing memory configuration since more compute is assigned. For the native *KDE* implementation (see Figure 5a), the Skylake processor obtains a speedup of 1.10x, 1.03x, on average across all memory configurations as compared to the Haswell and Broadwell processors. On the other hand, for the Numba optimized function, we observe an average speedup of 1.79x, 1.36x for the Skylake processor (with AVX-512) as compared to the Haswell and Broadwell processors respectively. We observe a difference in performance for the different architectures. This is because of several microarchitectural improvements to the Skylake processor [18]. The difference in performance is more significant

Authorized licensed use limited to: University of Washington Libraries. Downloaded on November 07,2023 at 08:00:59 UTC from IEEE Xplore. Restrictions apply.



(a) Kde Native. (b) Kde Numba

Fig. 5: Comparison of the execution times for the optimized and native versions of the *Kde* FaaS workload for the different underlying processor architectures. The functions were deployed on the `asia-northeast1` region.

for the Numba optimized function because the LLVM compiler in Numba autovectorizes the jitted function in the *KDE* workload to generate instructions using the AVX-512 instruction set on the Skylake processor and using the AVX-2 instruction set on the Haswell and Broadwell processors. As a sanity check, we also confirmed this by examining the assembly code of the jitted function and checking the registers used in the generated vector instructions. The Broadwell processor obtains a speedup of 1.03x, 1.31x on average across all memory configurations as compared to the Haswell processor for the native and Numba optimized functions respectively. This can be attributed to a higher Instructions per cycle (IPC) value and reduced latency for floating point operations as compared to the Haswell processor [20].

In comparison to the Numba optimized function with SSE and AVX-2 generated instructions on the Skylake processor, the version with AVX-512 instructions obtains a best speedup of 1.67x and 1.16x on average across all memory configurations respectively. Moreover, the SSE version on the Skylake processor is 1.23x slower on average than the optimized version with AVX-2 instructions on the Broadwell processor. Although there is an illusion of homogeneity in most public FaaS offerings, the actual performance of a FaaS function can vary depending on the underlying architecture of the provisioned VM where the function instance is launched. As a result, the cost incurred for the same function will also vary.

VI. Conclusion & Future Work

In this paper, we adapted and optimized three compute-intensive FaaS workloads with Numba, a JIT compiler based on LLVM. We determined the different processor architectures used by GCF namely Haswell, Broadwell, and Skylake in the underlying provisioned VMs on which the function instances are launched. Furthermore, we identified their prevalence across the 19 available GCF regions. Moreover, we demonstrated the use of the underlying VM configuration, i.e., number of vCPUs for parallelizing FaaS functions. We found that the performance of a particular optimized FaaS function can vary by 1.79x, 1.36x on average depending on the underlying processor. Moreover, under-optimization of a function based on the underlying architecture can degrade the performance by a value of 1.67x. In the future, we plan to investigate strategies for caching the compiled optimized machine code to reduce the startup times of functions.

VII. Acknowledgement

This work was supported by the funding of the German Federal Ministry of Education and Research (BMBF) in the scope of the Software Campus program. Google Cloud credits were provided by the Google Cloud Platform research credits.

References

[1] M. Chadha, A. Jindal, and M. Gerndt, "Towards federated learning using faas fabric," in *Proceedings of the 2020 Sixth International Workshop on Serverless Computing*, ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 49–54.

[2] E. Jonas, Q. Pu, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the cloud: Distributed computing for the 99%," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 445–451.

[3] R. Chard, Y. Babuji, Z. Li, T. Skluzacek, A. Woodard, B. Blaiszik, I. Foster, and K. Chard, "Funcx: A federated function serving fabric for science," in *Proceedings of the 29th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 65–76. [Online]. Available: https://doi.org/10.1145/3369583.3392683

[4] J. Kim and K. Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[5] A. Jindal, M. Gerndt, M. Chadha, V. Podolskiy, and P. Chen, "Function delivery network: Extending serverless computing for heterogeneous platforms," *Software: Practice and Experience*.

[6] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, 2018, pp. 133–146.

[7] D. Kelly, F. Glavin, and E. Barrett, "Serverless computing: Behind the scenes of major platforms," in *2020 IEEE 13th International Conference on Cloud Computing (CLOUD)*, 2020, pp. 304–312.

[8] J. Spillner, "Resource management for cloud functions with memory tracing, profiling and autotuning," ser. WoSC'20. New York, NY, USA: Association for Computing Machinery, 2020, p. 13–18. [Online]. Available: https://doi.org/10.1145/3429880.3430094

[9] Google Cloud Functions Pricing, https://cloud.google.com/functions/pricing, accessed 09/24/2020.

[10] A. Quach and A. Prakash, "Bloat factors and binary specialization," in *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation*, ser. FEAST'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 31–38. [Online]. Available: https://doi.org/10.1145/3338502.3359765

[11] S. K. Lam, A. Pitrou, and S. Seibert, "Numba: A llvm-based python jit compiler," in *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC*, ser. LLVM '15. New York, NY, USA: Association for Computing Machinery, 2015. [Online]. Available: https://doi.org/10.1145/2833157.2833162

[12] C. Lattner and V. Adve, "Llvm: a compilation framework for lifelong program analysis transformation," in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[13] The Python Benchmark Suite, https://github.com/python/pyperformance, accessed on 09/24/2020.

[14] A. Mohan, H. Sane, K. Doshi, S. Edupuganti, N. Nayak, and V. Sukhomlinov, "Agile cold starts for scalable serverless," in *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*. Renton, WA: USENIX Association, Jul. 2019. [Online]. Available: https://www.usenix.org/conference/hotcloud19/presentation/mohan

[15] A. Fuerst and P. Sharma, "Faascache: Keeping serverless computing alive with greedy-dual caching," 2021.

[16] GCF Locations, https://cloud.google.com/functions/docs/locations, accessed on 09/24/2020.

[17] Intel CPUs, https://en.wikichip.org/wiki/intel/cpuid, accessed on 09/24/2020.

[18] R. Schöne, T. Ilsche, M. Bielert, A. Gocht, and D. Hackenberg, "Energy efficiency features of the intel skylake-sp processor and their impact on performance," in *2019 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 2019, pp. 399–406.

[19] Controlling Scaling Behavior, https://cloud.google.com/functions/docs/max-instances, accessed on 09/24/2020.

[20] M. K. Kumashikar, S. G. Bendi, S. Nimmagadda, A. J. Deka, and A. Agarwal, "14nm broadwell xeon® processor family: Design methodologies and optimizations," in *2017 IEEE Asian Solid-State Circuits Conference (A-SSCC)*, 2017, pp. 17–20.

Authorized licensed use limited to: University of Washington Libraries. Downloaded on November 07,2023 at 08:00:59 UTC from IEEE Xplore. Restrictions apply.