Carbon Aware Serverless System

Jaykumar Patel
patel.jay4802@utexas.edu
jnp2369

Afnan Mir
afnanmir@utexas.edu

Nidhi Dubagunta
nidhi.dubagunta@utexas.edu
nsd632

Abstract

Serverless computing is a method that enables developers to build and deploy applications without having to provision resources or manage infrastructure. However, existing serverless systems do not consider the carbon footprint of function invocations when allocating or configuring resources, such as memory and vCPU. In this paper, we propose a carbon-aware serverless system that focuses on minimizing the carbon emissions of function invocations while continuing to meet service level objectives (SLOs) by finding an optimal configuration for vCPU count and operational frequency of the vCPU cores. We first perform a measurement study by running a variety of workloads with different resource configurations to understand the relationship between resource usage, energy consumption, and performance. We also collect measurements for energy consumption during different stages in the serverless function invocation lifecycle. Leveraging these insights, we aim to incorporate a Bayesian Optimization algorithm into Openwhisk, an open-source and serverless cloud platform, to build a serverless system that performs carbon-aware dynamic frequency scaling (DFS) and resource allocation.

1 Introduction

The rise of cloud computing has transformed the space of digital infrastructure, offering higher amounts of scalability, flexibility, and cost-effectiveness compared to on-premise solutions. Within cloud computing, the concept of serverless computing emerged, which further increased efficiency in application development by abstracting the complexities of infrastructure management and resource provisioning.

In a serverless system, when a function is invoked, the system checks for a pre-existing warmed container to host the function. If no warmed container is available, a cold start is initiated, during which the runtime environment is set up, necessary dependencies are installed, and the function code is loaded. After the function is executed, the container can remain idle for a period of time for future invocations before it is killed.

Several existing serverless systems, such as AWS and GCP, couple memory and CPU allocation, assigning a CPU share proportional to the memory requested by the user, or using a preset resource allocation option. https://mcanini.github.io/papers/serverless.eurosys23.pdf serverless systems today also do not incorporate dynamic frequency scaling (DFS), in which the operating frequency of the vCPU cores can be dynamically modified. These policies used for resource allocation are carbon-agnostic, meaning they do not consider the carbon emissions of the function execution and idle containers.

In this paper, we propose a serverless framework that provisions and configures resources based on workload characteristics to minimize the carbon emissions associated with the function invocation, while continuing to meet SLOs, such as latency, throughput, total runtime, etc. The features we aim to optimize are the number of CPUs allocated, as well as the frequency at which the CPU cores operate.

2 Motivation

In a study published by Microsoft in collaboration with the Green Software Foundation, they state that data centers and data transmission networks account for almost 1% of energy-related greenhouse gas emissions, which contribute to rising global temperatures and climate change. By adopting carbon-aware computing, these data centers can play a significant role in decarbonization efforts.

Carbon-aware computing not only places emphasis on minimizing carbon emissions associated with deploying and maintaining cloud infrastructures (or in our case serverless functions), but also on optimizing hardware utilization.

Through optimizing the amount of vCPU allocated and the frequency of the cores for each specific function invocation, we can both work to minimize the total energy expenditure and therefore carbon footprint, as well encourage

high resource utilization, by catering to the specific performance patterns of the function.

3 Related Work

CherryPick is a framework that was developed in order to find optimal or near-optimal cloud configurations (vCPU and memory) for big data analytics jobs, addressing challenges of system cost, performance, and adaptivity. Cherrypick leverages Bayseian Optimization to build performance models that are able to determine optimal or near-optimal configurations that satisfy performance constraints with a relatively low number of samples. Our work aims to draw from the machine learning techniques utilized in the paper and extend it to the application of finding the near-optimal configuration of vCPU and operational frequency with the objective of minimizing carbon footprint. Our work will also set performance constraints to ensure user-specified SLOs are met.

Another work that utilizes Bayesian models to optimize for resource allocation is Aquatope. This framework also utilizes Bayesian optimization to learn the mapping from resource configuration to performance and cost, while considering noise and uncertainty in the cloud environment. Aquatope also uses batch sampling to reduce the cost of exploration when finding a resource configuration, (which could potentially be integrated into our work as well ??).

In the space of carbon-aware computing, GreenCourier is a scheduling framework that enables runtime scheduling of serverless functions across geographically distributed regions based on their carbon efficiencies. This work focuses on minimizing carbon footprint based on the location at which a function is run, whereas our work aims to optimize the host environment through proper resource allocation and configuration to minimize carbon emissions using function-level characteristics.

4 Measuring Carbon Emission

Before we conduct the measurement study, we need to define how to measure carbon emissions. According to Gupta et al. [1], the carbon footprint of a server can be calculated using the following formula where OP_{CF} is the operational carbon footprint, CI_{use} is the carbon intensity of the energy source, and $Energy$ is the operational energy:

$$OP_{CF} = CI_{use} \times Energy \quad (1)$$

The carbon intensity refers to how *clean* the energy source is. For example, coal has a higher carbon intensity than solar. We assume that the energy source of the serverless system is constant, and therefore, the carbon intensity is constant. We can then use the operational energy as a proxy for the carbon footprint.

4.1 Measuring Energy

We first need a way to measure the energy consumption of a serverless function invocation. EnergAt [2] is a tool that measures fine-grained energy consumption at the thread-level while taking into account NUMA effects. An EnergAt process continuously samples a target container, where each sample returns energy readings over an interval of at least 50ms. Therefore, EnergAt will get continuous back-to-back samples of energy readings over time. The energy consumption of a target container is the sum of the energy readings over the duration of the function invocation. One limitation of EnergAt is that every target container for which we want to measure energy consumption must be instrumented with its own EnergAt process. This is not feasible as the number of EnergAt processes and resource contention scales with the number of target containers.

To mitigate this issue, we created EnergyLiteDaemon. EnergyLiteDaemon is a daemon that samples multiple target containers in a round-robin fashion, where each sample returns energy readings over an interval of at least 50ms. This allows us to measure energy consumption of multiple target containers with a single, light-weight, process. Due to the round-robin fashion, however, the sampling frequency per target container decreases as a function of the number of target containers. If there are multiple containers, the sample are no longer back-to-back – there may be some time delay between two samples. To solve this, we complete the following steps to interpolate the total energy consumption of a target container:

Power Calculation: We compute the average power for every sample by dividing the energy consumed over the interval of the sample by the duration of that interval. This allows us to get a power-time which contains n points, where n is the number of samples.

Trapezoidal Integration: Given a power-time scatterplot, we compute the total energy consumption of a target container by performing trapezoidal integration. Essentially, all of the points in the power-time scatter-plot are connected by straight lines, and the area under the curve is calculated. This area is the total energy consumption of the target container.

A toy example of energy measurement using EnergAt and EnergyLiteDaemon is shown in Appendix A.1.

Due to EnergyLiteDaemon's round-robin sampling approach, which reduces its overhead, it may not capture all energy readings for a container. Even with trapezoidal integration, the energy consumption will not be exact. To determine the accuracy of EnergyLiteDaemon, we ran 20 stress docker containers, with varying CPU usage and duration. Figure 1 shows the energy consumption of the containers as measured by EnergAt and EnergyLiteDaemon. The absolute percentage error of EnergyLiteDaemon is less than

Variable	Values
Function Type	Float Matrix Multiplication, Image Processing, Video Processing, Encryption, Linpack
vCPU Count	1, 2, 3, 5, 7, 10, 13, 16, 19, 22, 25, 28, 31
vCPU Frequency (GHz)	1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2. 2.4

Table 1: Measurement Study Configurations – Energy Variation with vCPU and Frequency Allocation

15%, which is acceptable for our purposes.

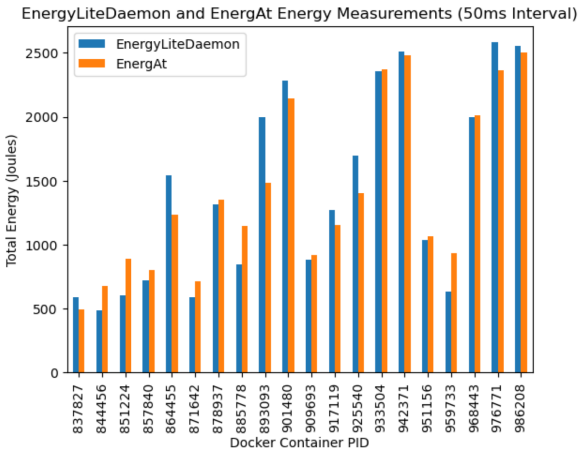


Figure 1: Energy measurements of EnergAt and EnergyLiteDaemon for 20 stress docker containers.

5 Measurment Study

We conduct two measurement studies to answer the following questions: 1. How does energy vary with changes in vCPU and frequency allocation? 2. What is the energy consumption over the lifecycle of a serverless function invocaiton?

To perform the measurement study, we use OpenWhisk, which is an open-source serverless platform, to run function invocations. We use EnergyLiteDaemon to measure the energy consumption of the function invocations.

5.1 Energy Variation with vCPU and Frequency Allocation

We ran a variety of workloads on a variety of vCPU and frequency configurations. We collected the vCPU utilization, energy consumption, and performance in terms of duration. The resource configurations are shown in Table 1.

We analyzed the data to understand how energy consumption varies with changes in vCPU and frequency allocation. We found that for functions that are highly parallelizable, such as float matrix multiplication, energy consumption decreases with the number of vCPUs, since the function duration decreases drastically. For functions that are not parallelizable, such as encryption, vCPU allocation does not affect energy consumption. For both types of functions, energy consumption as a function of frequency displays a convex curve, where energy consumption decreases with frequency until a certain point, and then increases with frequency. This is because the energy consumption of a vCPU is proportional to the frequency at which it runs, but the duration of the function is inversely proportional to the frequency at which the vCPU runs. More details are in Appendix A.2.

These trends support the idea that we can use machine learning to learn the relationship between vCPU, frequency, and energy consumption, and use this model to minimize energy consumption while meeting SLOs.

5.2 Energy Consumption Over the Lifecycle of a Serverless Function Invocation

Additionally, we ran a variety of docker images, each of a different size, with a variety of vCPU and memory configurations. For each (image, vCPU, memory) configuration, we measured the energy consumption at different stages of the function invocation lifecycle. More specifically, we measured the energy consumed during the container spin-up, the idle time, and the container spin-down.

After collecting and analyzing the data, we first found that for each stage of the lifecycle we measured, the amount of vCPU and memory configurations, the energy consumption did not vary significantly. This was true for all image sizes. This is likely due to the fact that these resources will not actually be allocated to the container until the compute inside the container actually requires it. We did notice that if we varied the image size, the energy consumption of the container spin-up process significantly increased as the image size increased. However, it is worth noting that most of the energy consumption of the spin-up is accredited to the 'docker pull' process, which would only be a one-time operation. The 'docker run' does not consume a significant amount of energy and does not show any statistical correlation. For the idle time, we noticed a strange negative correlation trend with the image size, and for the spin-down, we noticed a somewhat positive correlation with a few outlier points. Because we only used a few image sizes, we cannot definitively conclude that these trends will extrapolate.

Due to our findings of little to no correlations found, we decided to not pursue a carbon-aware keep-alive policy further. More details on the study can be found in Appendix A.3.

6 Next Steps

6.1 Machine Learning Model

6.2 Adapting OpenWhisk

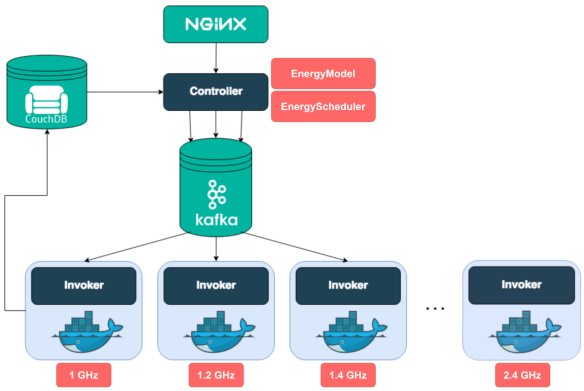


Figure 2: Adapted OpenWhisk System Overview

We will adapt OpenWhisk to incorporate Bayesian Optimization to drive a carbon-aware DFS and vCPU allocation model. Figure 2 shows the system overview.

The Bayesian Optimization model will be contained in a shim layer, called the EnergyModel. Whenever a user invokes a function, the EnergyModel is be used to determine the optimal vCPU and frequency allocation (using Bayesian Optimization). The EnergyModel will be trained on the data collected from the measurement study.

We will also create a custom scheduler, called EnergyScheduler, that will replace OpenWhisk's current scheduler. The EnergyScheduler will use the output of EnergyModel to determine the optimal vCPU to allocate for a function invocation. There will also be multiple invoker nodes, each running at a different frequency. The EnergyScheduler will send the request to the invoker that is running at the frequency predicted by EnergyModel.

References

[1] U. Gupta, M. Elgamal, G. Hills, G.-Y. Wei, H.-H. S. Lee, D. Brooks, and C.-J. Wu. Act: designing sustainable computer systems with an architectural carbon modeling tool. In *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ISCA '22, page 784–799, New York, NY, USA, 2022. Association for Computing Machinery.

[2] H. Hè, M. Friedman, and T. Rekatsinas. Energat: Fine-grained energy attribution for multi-tenancy. In *Proceedings of the 2nd Workshop on Sustainable Computer Systems*, HotCarbon '23. ACM, July 2023.

A Appendix

A.1 Energy Measurement

A toy example of the energy measurement using EnergAt and EnergyLiteDaemon is shown in Figure 3. Figure 3a shows the energy readings of EnergAt. Figure 3b shows the energy readings of EnergyLiteDaemon for one container. EnergyLiteDaemon doesn't capture all energy readings for a container due to the round-robin sampling approach. Figure 3c shows EnergyLiteDaemon's power calculation of of each energy sample. Figure 3d shows the energy interpolation of EnergyLiteDaemon using trapezoidal integration.

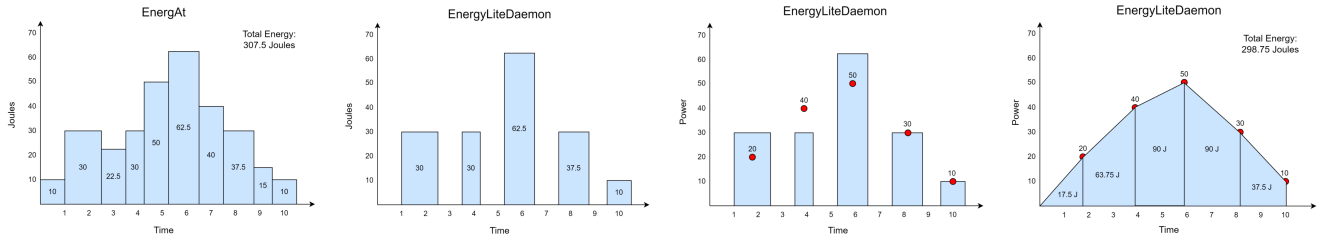
A.2 Energy Variation with vCPU and Frequency Allocation

The energy usage, duration, and CPU usage for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU allocation are shown in Figure 4. For Float Matrix Multiplication, energy usage initially decreases and then flat-lines as the vCPU allocation increases. This is because matrix multiplication is highly parallelizable and results in higher vCPU utilization, leading to a decrease in duration and energy usage. For Image Processing, energy usage has no apparent correlation with vCPU allocation. This is because most of image processing tasks are non-parallelizable. For Encryption, energy usage, duration, and vCPU utilization do not change with vCPU allocation. This is because encryption is non-parallelizable and does not benefit from additional vCPUs.

The energy usage and duration for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU frequency are shown in Figure 5. For all functions, energy usage displays a convex curve as a function of frequency. This is because the energy consumption of a vCPU is proportional to the frequency at which it runs, but the duration of the function is inversely proportional to the frequency at which the vCPU runs. Therefore, energy usage decreases with frequency until a certain point, and then increases with frequency.

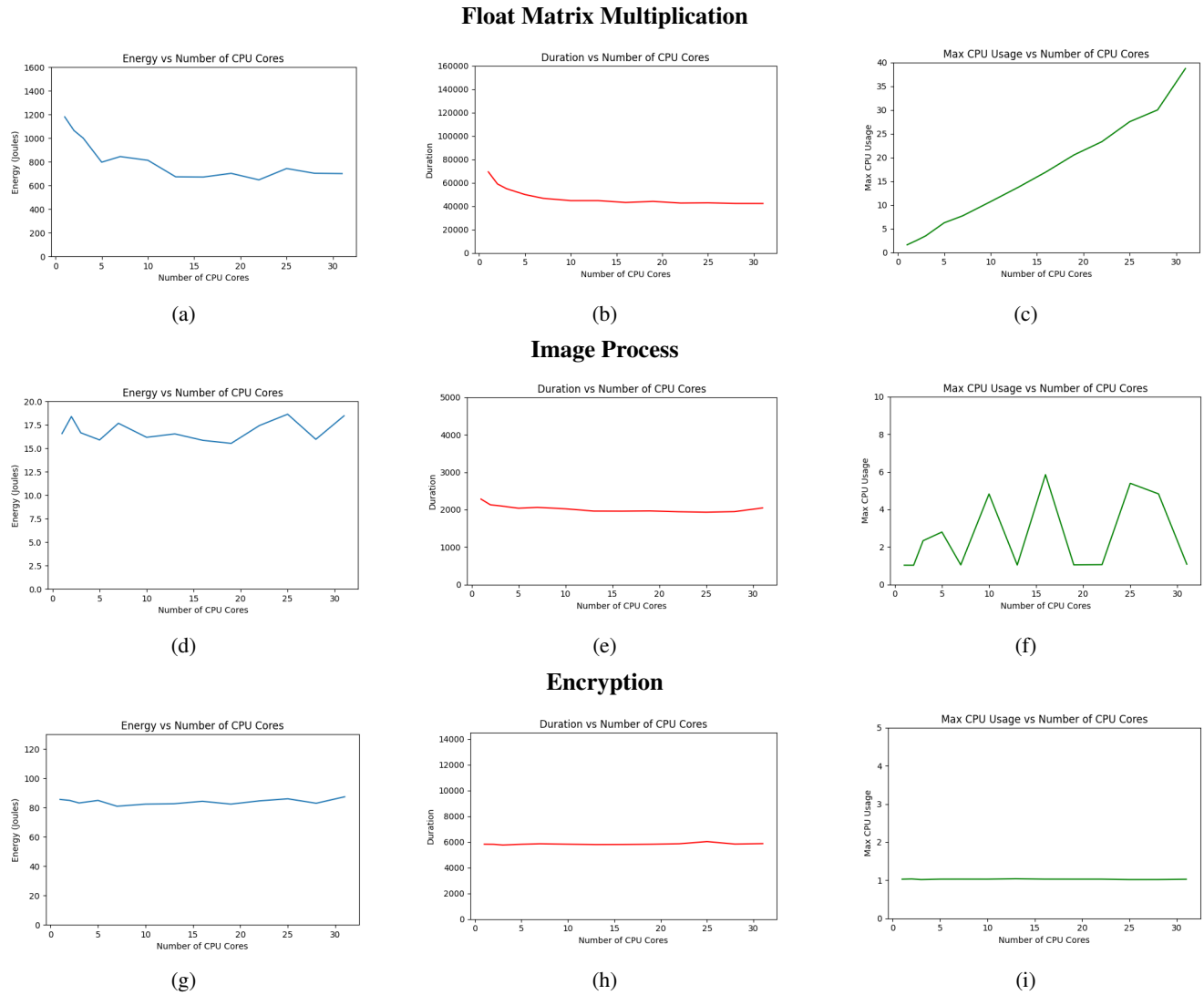
A.3 Energy Variation for Different Stages of a Serverless Function Invocation

We can see here the energy consumption of the container spin-up, idle time, and container spin-down for different image sizes in Figure 6.



(a) (b) (c) (d)

Figure 3: Toy example of energy measurement using EnergAt and EnergyLiteDaemon. (a) EnergAt energy readings. (b) EnergyLiteDaemon energy readings for one container. (c) EnergyLiteDaemon power calculation (red dots). (d) EnergyLiteDaemon energy interpolation using trapezoidal integration.



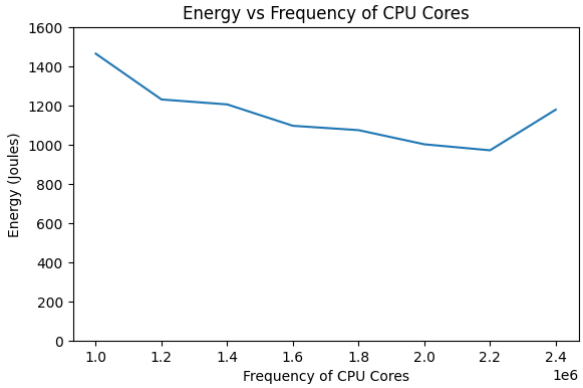
(a) (b) (c)

(d) (e) (f)

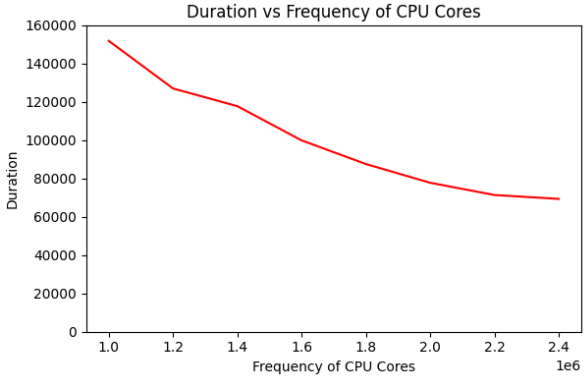
(g) (h) (i)

Figure 4: Energy Usage (a), Duration (b), and CPU Usage (c) for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU allocation. Frequency was fixed at 2.4 GHz.

Float Matrix Multiplication

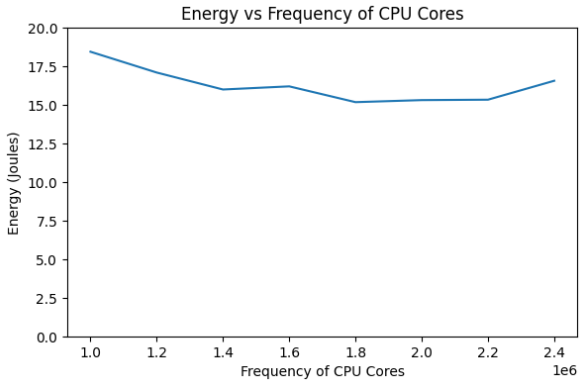


(a)

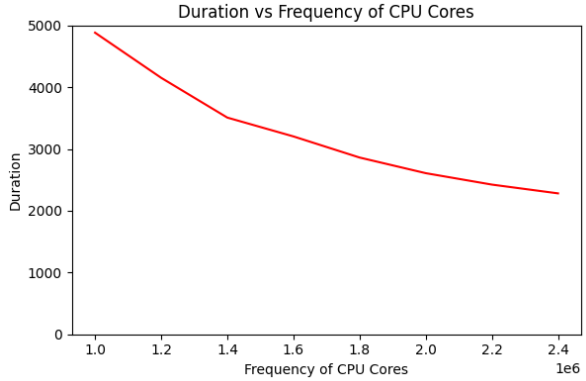


(b)

Image Process

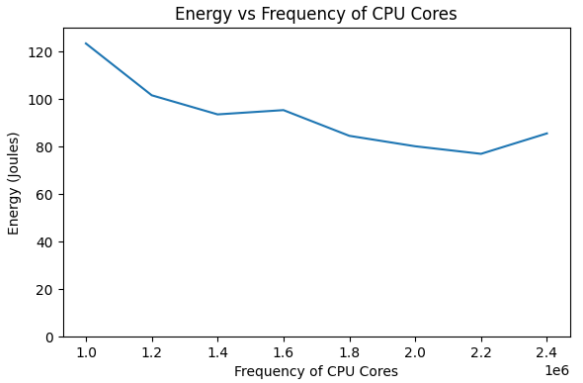


(c)

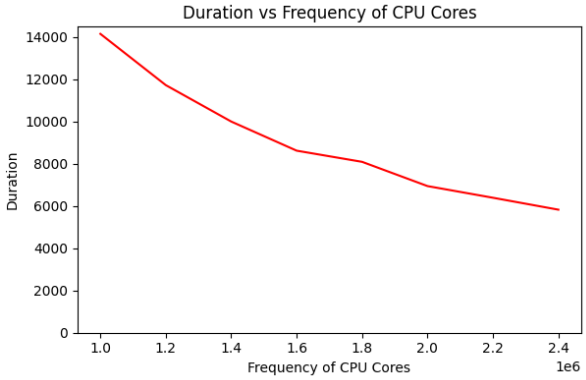


(d)

Encryption



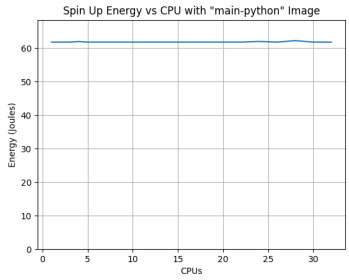
(e)



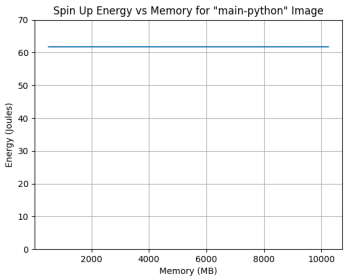
(f)

Figure 5: Energy Usage (a), Duration (b), and CPU Usage (c) for Float Matrix Multiplication, Image Processing, and Encryption as a function of vCPU frequency. vCPU allocation was fixed at 1.

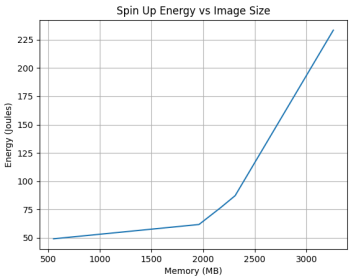
Spin Up Stage



(a)

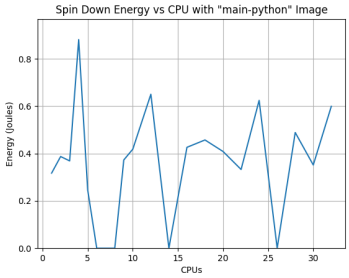


(b)

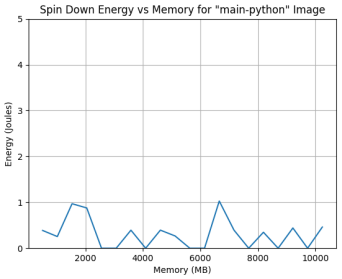


(c)

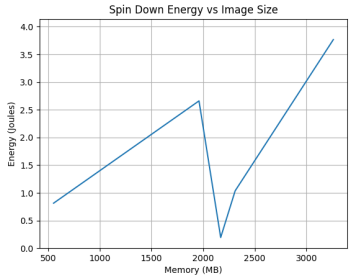
Spin Down Stage



(d)

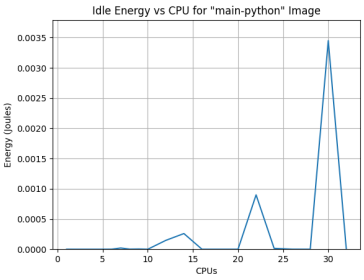


(e)

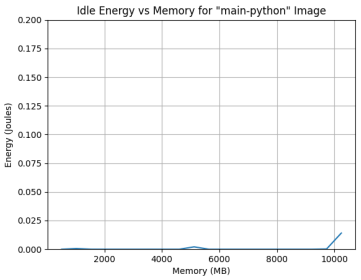


(f)

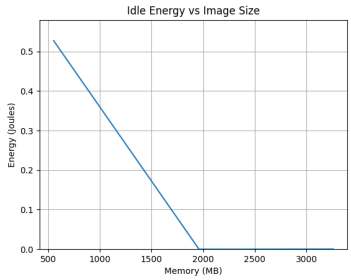
Idle Stage



(g)



(h)



(i)

Figure 6: Energy Usage vs CPU (a, d, g), Memory (b, e, h), and Image Size (c, f, i) for the Spin Up, Spin Down, and Idle stages of a serverless function invocation.