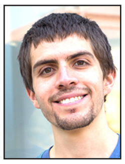
Outsourcing Everyday Jobs to Thousands of Cloud Functions with gg

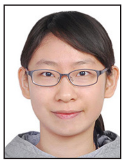
SADJAD FOULADI, FRANCISCO ROMERO, DAN ITER, QIAN LI, ALEX OZDEMIR, SHUVO CHATTERJEE, MATEI ZAHARIA, CHRISTOS KOZYRAKIS, AND KEITH WINSTEIN

Sadjad Fouladi is a PhD candidate in computer science at Stanford University, working with Keith Winstein on topics in networking, video systems, and distributed computing. His current projects include general-purpose lambda computing and massively parallel ray-tracing systems. sadjad@cs.stanford.edu



Francisco Romero is a PhD student in electrical engineering at Stanford University. His interests are in computer architecture and computer systems. He has recently worked on in-memory database systems for emerging storage technologies, serverless computing, machine learning inference systems, and datacenter resource scheduling. faromero@stanford.edu

Dan Iter is a PhD student at Stanford University. He is advised by Professor Dan Jurafsky and is a member of the NLP Group and AI Lab. He is interested in generative models for text representation, relation extraction, knowledge-base construction, and mental health applications. Previously, Dan also worked on lambda computing and virtualized storage for datacenters. daniter@stanford.edu



Qian Li is a PhD student in computer science at Stanford University, advised by Professor Christos Kozyrakis. She has broad interests in computer systems and architecture. Her current research focuses on efficient resource management and scheduling for heterogeneous cloud computing platforms. Before coming to Stanford, Qian received her Bachelor of Science from Peking University. qianli@cs.stanford.edu

We introduce `gg`, a framework that helps people execute everyday applications—software compilation, unit tests, video encoding, or object recognition—using thousands of parallel threads on a "serverless" platform to achieve near-interactive completion times. We envision a future where instead of running these tasks on a laptop, or keeping a warm cluster running in the cloud, users push a button that spawns 10,000 parallel cloud functions to execute a large job in a few seconds from start. `gg` is designed to make this practical and easy.

A third of a century ago, interactive personal computing changed the way the computers were used and markedly increased global productivity. Nevertheless, even today, many applications remain far from interactive: compiling a large software package can take hours; processing an hour of 4K video typically needs more than 30 CPU-hours; and a single frame from the animated movie *Monsters University* takes 29 hours to render [8]. Users who wants to explore or tinker and desire feedback in seconds need to harness thousands of cores in parallel, far exceeding the available compute power in laptops and workstations and leading users towards rented compute resources in large-scale datacenters—the cloud.

However, outsourcing a job to thousands of threads in the cloud presents its own challenges. For one, maintaining a warm cluster of thousands of CPU cores in the form of VMs is not cost-effective for occasional short-lived jobs. Provisioning and booting a cluster of VMs on current commercial services can also take several minutes, leaving end users with no practical option to scale their resource footprint *on demand* in an efficient and scalable manner.

Meanwhile, a new category of cloud-computing resources has emerged that offers finer granularity and lower latency than traditional VMs: cloud functions, also called *serverless computing*. Amazon's Lambda service will rent a Linux container for a minimum of 100 ms, with a startup time of less than a second and no charge when idle. Google, Microsoft, Alibaba, and IBM have similar offerings.

Cloud functions were intended for asynchronously invoked microservices, but their granularity and scale sparked our interest for a different use: as a burstable supercomputer-on-demand. As part of building our massively parallel, low-latency video-processing system, ExCamera [4], we found that thousands of cloud functions can be invoked in a few seconds with inter-function communication over TCP, effectively providing something like a rented 10,000-core computer billed by the second. ExCamera's unorthodox use of a cloud-functions service has been followed by several subsequent systems, including PyWren [6], Sprocket [2], Cirrus, Serverless MapReduce, and Spark-on-Lambda. These systems all launch a *burst-parallel* swarm of thousands of cloud functions, all working on the same job, to provide results to an interactive user.

Challenges of Building Burst-Parallel Applications

Despite the above, building new burst-parallel applications on thousands of cloud functions has remained a difficult task. Each application must overcome a number of challenges endemic to this environment: (1) workers are stateless and may need to download large

Alex Ozdemir is a PhD student in computer science at Stanford University. His research interests span much of theoretical computer science and computer systems. aozdemir@stanford.edu

Shuvo Chatterjee currently works at Google on account security. Previously, he worked at Square and Apple. In between, he was a visiting researcher at Stanford. His focus is primarily on user security and privacy in large-scale systems. He is a graduate of MIT. shuvo@alum.mit.edu

Matei Zaharia is an Assistant Professor of Computer Science at Stanford University and Chief Technologist at Databricks. He works on computer systems for data analysis, machine learning, and security as part of the Stanford DAWN lab. Previously, Matei started the Apache Spark project during his PhD at UC Berkeley in 2009 and co-started the Apache Mesos cluster manager. Matei's research work was recognized through the 2014 ACM Doctoral Dissertation Award for the best PhD dissertation in computer science, an NSF CAREER Award, the VMware Systems Research Award, and best paper awards at several conferences. matei@cs.stanford.edu

Christos Kozyrakis is a Professor in the Departments of Electrical Engineering and Computer Science at Stanford University. His research interests include resource-efficient cloud computing, energy-efficient computing and memory systems for emerging workloads, and scalable operating systems. Kozyrakis has a PhD in computer science from the University of California, Berkeley. He is a Fellow of the IEEE and ACM. kozyraki@stanford.edu

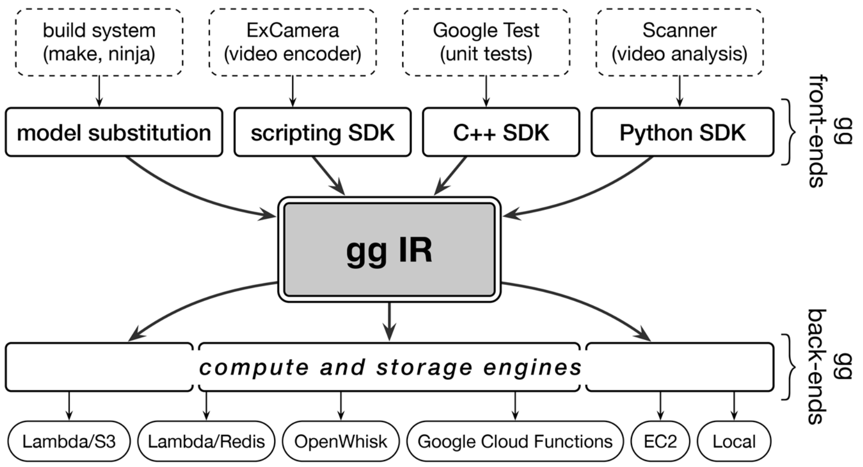


Figure 1: gg helps applications express their jobs in an intermediate representation that abstract the application logic from its placement, schedule, and execution, and provides back-end engines to execute the job on different cloud-computing platforms.

amounts of code and data on startup; (2) workers have limited runtime before they are killed; (3) on-worker storage is limited but much faster than off-worker storage; (4) the number of available cloud workers depends on the provider's overall load and can't be known precisely upfront; (5) worker failures are more likely to occur when running at large scale; (6) libraries and dependencies differ in a cloud function compared with a local machine; and (7) latency to the cloud makes roundtrips costly.

In this article, we present gg, a general system designed to help application developers manage the challenges of creating burst-parallel cloud-function applications. Instead of directly targeting a cloud-functions infrastructure, application developers express their jobs in gg's *intermediate representation* (gg IR), which abstracts the application logic from its placement, schedule, and execution. This portable representation allows gg to run the same application on a variety of compute and storage platforms, and provides runtime features that address underlying challenges, such as dependency management, straggler mitigation, placement, and memoization. Figure 1 illustrates the overall architecture of gg.

gg can containerize and execute existing programs, e.g., software compilation, unit tests, video encoding, or searching a movie with an object-recognition kernel. gg does this with thousands-way parallelism on short-lived cloud functions. In some cases, this yields considerable benefits in terms of performance. For example, compiling the Inkscape graphics editor on AWS Lambda using gg was almost 5x faster than an existing system (icecc) running on a 384-core cluster of warm VMs.

```
{ function: {
    hash: 'VDSo_TM',
    args: [ 'gcc', -E', 'hello.c', '-o', 'hello.i' ],
    envars: [ 'LANG=us_US' ] },
  objects: [ 'VLb1SuN=hello.c', 'VDSo_TM=gcc', 'VAs.BnH=cpp', 'VB33fCB=/usr/stdio.h' ],
  outputs: [ 'hello.i' ] }
```
thunk hash: **T0MEiRL**

Figure 2: An example thunk for preprocessing a C program, hello.c. The thunk is named by the hash of its content, T0MEiRL. The hash starts with T to mark it as a thunk rather than a primitive value. Other thunks can refer to its output by using this hash.

Keith Winstein is an Assistant Professor of Computer Science (and, by courtesy, of Electrical Engineering) at Stanford University. He and his students and colleagues made the Mosh (mobile shell) tool, the Mahimahi network emulator, the Sprout and Remy systems for computer-generated congestion control, the Lepton functional-compression tool used at Dropbox, the ExCamera, Salsify, and Puffer systems for video coding and transmission, the Pantheon of Congestion Control, and gg. keithw@cs.stanford.edu

Thunks: Transient Functional Containers

The heart of gg IR is an abstraction that we call a *thunk*. In the functional-programming literature, a thunk is a parameterless closure that captures a snapshot of its arguments and environment for later evaluation. The process of evaluating the thunk—applying the function to its arguments and saving the result—is called *forcing* it [1].

Building on this concept, gg represents a thunk with a description of a container that identifies, in content-addressed manner, an x86-64 Linux executable and all of its input data objects. The container is hermetically sealed and meant to be referentially transparent; it is not allowed to use the network or access unlisted objects or files. The thunk also describes the arguments and environment for the executable and a list of tagged output files that it will generate—the results of forcing the thunk. Figure 2 shows an example thunk for preprocessing a C source file. Since the thunk captures the full functional footprint of a function, it can be executed in any environment capable of running an x86-64 Linux executable.

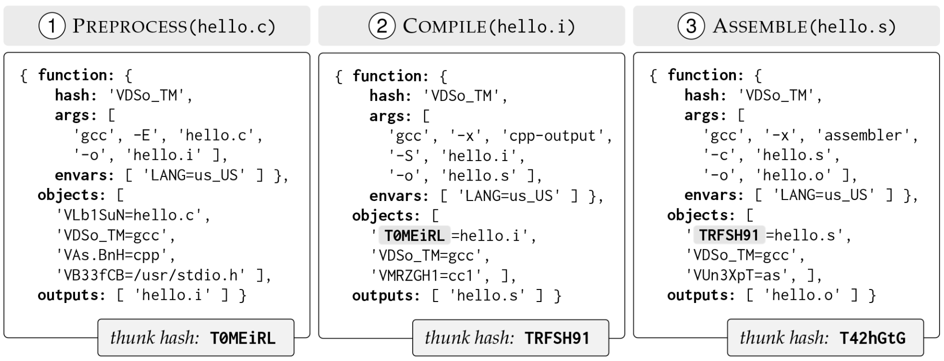
All the objects, including the input files, functions, and thunks are named by their hashes. More precisely, the name of an object has four components: (1) whether the object is a primitive value (hash starting with V) or refers to the result of forcing some other thunk (hash starting with T), (2) a SHA-256 hash of the value's or thunk's content, (3) the length in bytes, and (4) an optional tag that names an object or a thunk's output.

Because the objects are content-addressed, they can be stored on any mechanism capable of producing a blob that has the correct name: durable or ephemeral storage (e.g., S3, Redis, or Bigtable), a network transfer from another node, or by finding the object already available in RAM from a previous execution.

From our experiences of working with the system, we expect gg thunks to be simple to implement and reason about, straightforward to execute, and well matched to the statelessness and unreliability of cloud functions.

gg IR: A Lazily Evaluated Lambda Expression

The structure of interdependent thunks—essentially a lambda expression—is what defines the gg IR. This representation exposes the computation graph to the execution engine, along with the identities and sizes of objects that need to be communicated between thunks. For example, the IR representing the expression `Assemble(Compile(Preprocess(hello.c)))` consists of three thunks, as depicted in Figure 3. Each stage refers to the previous stage's output by using the thunk's hash.



```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', -E', 'hello.c',
      '-o', 'hello.i' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    'VLb1SuN=hello.c',
    'VDSo_TM=gcc',
    'VAs.BnH=cpp',
    'VB33fCB=/usr/stdio.h' ],
  outputs: [ 'hello.i' ] }
```

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'cpp-output',
      '-S', 'hello.i',
      '-o', 'hello.s' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    ' T0MEiRL =hello.i',
    'VDSo_TM=gcc',
    'VMRZGH1=cc1', ],
  outputs: [ 'hello.s' ] }
```

```
{ function: {
    hash: 'VDSo_TM',
    args: [
      'gcc', '-x', 'assembler',
      '-c', 'hello.s',
      '-o', 'hello.o' ],
    envars: [ 'LANG=us_US' ] },
  objects: [
    ' TRFSH91 =hello.s',
    'VDSo_TM=gcc',
    'VUn3XpT=as', ],
  outputs: [ 'hello.o' ] }
```

Figure 3: An example of gg IR consisting of three thunks for building a "Hello, World!" program that represents the expression `Assemble(Compile(Preprocess(hello.c)))` → `hello.o`. To produce the final output `hello.o`, thunks must be forced in order from left to right. Other thunks, such as the link operation, can reference the last thunk's output using its hash, `T42hGtG`. Hashes have been shortened for display.

The IR allows gg to schedule jobs efficiently, mitigate the effect of stragglers by invoking multiple concurrent thunks on the critical path, recover from failures by forcing a thunk a second time, and memoize thunks to avoid repetitive work. This is achieved in an application-agnostic, language-agnostic manner. Based on the data exposed by the IR, back ends can schedule the forcing of thunks, place thunks with similar data-dependencies or an output-input relationship on the same physical infrastructure, and manage the storage or transfer of intermediate results, without roundtrips back to the user's own computer.

Front-End Code Generators and Back-End Execution Engines

Front ends are the programs that emit gg IR (Figure 1). Most of the time, we expect the applications to write out thunks by explicitly providing the executable and its dependencies. This can be done through a command-line tool provided by gg (i.e., `gg create-thunk`) or by using the C++ and Python SDKs that expose a thunk abstraction and allow the developer to describe the application in terms of thunks. For one application, software compilation, we developed a technique called *model substitution* that is designed to extract gg IR from an *existing* build system, without actually compiling the software. In the next section, we will describe the details of this technique.

The execution of gg IR is done by the back ends and requires two components: an *execution engine* for forcing the individual thunks, and a content-addressed *storage engine* for storing the named blobs referenced or produced by the thunks. We implemented five compute engines (a local machine, a cluster of warm VMs, AWS Lambda, IBM Cloud Functions, and Google Cloud Functions) and three storage engines (S3, Google Cloud Storage, and Redis).

gg's approach of abstracting front ends from back ends allows the applications and the back-end engines to evolve and improve independently. The developers can focus on building new applications on top of gg abstractions and, at the same time, benefit from the improvements made to the execution back ends. Moreover, special-purpose execution engines can be built to match the unique characteristics of a certain job without changing the IR description of the application.

As an example, our default AWS Lambda/S3 back end invokes a new Lambda for each thunk. Upon completion, a Lambda uploads its outputs to S3 for other workers to download and use. However, for applications like ExCamera that deal with large input/output objects, the roundtrips to S3 can negatively affect the performance. To improve the performance of such applications, we made a "long-lived" AWS Lambda engine, where each worker stays up until the whole job finishes and seeks out new thunks to execute. The execution engine keeps an index of objects present on each worker's local storage and uses that information to place thunks on workers with the most data available, in order to minimize the need to fetch dependencies from the storage back end.

Software Compilation with gg

Software compilation has long been a prime example of non-interactive computing. For instance, compiling the Chromium Web browser, one of the largest open-source projects, takes more than four hours on a 4-core laptop. Many solutions have been developed to leverage warm machines in a local cluster or cloud datacenter (e.g., `distcc` or `icecc`). We developed such an application on top of gg that can outsource a compilation job to thousands of cloud functions.

Build systems are often large and complicated. The application developers have spent a considerable amount of time crafting `Makefiles`, `CMakeLists.txt` files, and `build.ninja` files for their projects, and manually converting them to gg IR is virtually impossible. We developed a technique called model substitution that can automatically extract a gg IR description from an existing build system.

We run the build system with a modified `PATH` so that each stage is replaced with a stub: a *model* program that understands the behavior of the underlying stage well enough so that when the model is invoked in place of the real stage, it can write out a thunk that captures the arguments and data that will be needed in the future; forcing the thunk will then produce the exact output that would have been produced during actual execution. We used this technique to infer gg IR from the existing build systems for several large open-source applications, including

	Local (make)		Distributed (icecc)	**Distributed (gg)**
	1 core	48 cores	48 cores	AWS Lambda
FFmpeg	06m 9s	20s	01m 03s	44s±04s
GIMP	06m 48s	49s	02m 35s	01m 38s±03s
Inkscape	32m 34s	01m 40s	06m 51s	01m 27s ±07s
Chromium	15h 58m 20s	38m 11s	46m 01s	18m 55s ±10s

Table 1: Comparison of cold-cache build times in different scenarios. gg on AWS Lambda is competitive with or faster than using conventional outsourcing (`icecc`) and, in the case of the largest programs, is 2–5× faster. This includes both the time required to generate gg IR from a given repository using model substitution and the time needed to execute the IR.

OpenSSH, the FFmpeg video system, the GIMP image editor, the Inkscape vector graphics editor, and the Chromium browser, with no changes to the original build system or user intervention. Table 1 shows a summary of the results for four open-source projects. gg on AWS Lambda is about 2–5× faster than a conventional tool (icecc) in building medium- and large-sized software packages.

As an example, we will go through the steps of building the FFmpeg video system with gg. First, the user clones the repo and execute ./configure script to generate the Makefiles:

```
sadjad@~$ git clone https://git.ffmpeg.org/ffmpeg.git
sadjad@~$ [install ffmpeg build dependencies]
sadjad@~$ cd ffmpeg
sadjad@~/ffmpeg$ ./configure --disable-doc --disable-x86asm
```

Next, the user runs gg init in the program's root, which will create a directory named gg. This directory will contain the generated thunks and local cache entries:

```
sadjad@~/ffmpeg$ gg init
```

To compile the project with gg, first we need to extract an IR description from the build system, which is done by running the normal build command (make in this case), prefixed by gg infer:

```
sadjad@~/ffmpeg$ gg infer *make -j$(nproc)*
```

This command will execute the underlying build system, but it modifies the PATH so that each stage of the build is replaced with a *model* program, which writes out a thunk for that stage. After the IR generation is done, the build targets are created, but their contents are not what we would normally expect:

```
sadjad@~/ffmpeg$ cat ffmpeg
#!/usr/bin/env gg-force-and-run
Te6aLo5FtpPyyGY.CsF8PHGY5WS61AlmbcUNGA1tG9Cs00000179
```

This is a *placeholder*, and it expresses that the actual ffmpeg binary is the output of the thunk with the hash Te6aLo5F... (the content of this thunk can be inspected by using the gg describe utility). Running this script forces this thunk, replaces itself with the output, and then executes it. The user can also manually force this thunk by using the gg force utility:

```
sadjad@~/ffmpeg$ gg force --jobs *1500* --engine *lambda*
ffmpeg
* Loading the thunks... done (233 ms).
* Uploading 4663 files (81.8 MiB)... done (6985 ms).
  ...
* Downloading output file (16.7 MiB)... done (1131 ms).
```

This command specifies that the user wants to run this job with 1500-way parallelism on AWS Lambda. First, all the necessary input files are uploaded to the storage engine in one shot. Then the program forces all the necessary thunks recursively until obtaining the final result. After the output is downloaded, the ffmpeg binary can be executed, as if it were built on the local machine:

```
sadjad@~/ffmpeg$ ./ffmpeg
ffmpeg version N-94028-gb8f1542dcb Copyright (c) 2000-2019
the FFmpeg developers
```

Unit Testing with gg

Software test suites are another set of applications that can benefit from massive parallelism, as each test is typically a standalone program that can be run in parallel with other tests, with no inter-dependencies. Using gg's C++ SDK, we implemented a tool that can generate gg IR for unit tests written with Google Test, a popular C++ test framework used by projects like LLVM, OpenCV, Chromium, Protocol Buffers, and the VPX video codec library.

For code bases with large numbers of test cases, this can yield major improvements. For example, the VPX video codec library contains more than 7,000 unit tests, which take more than 50 minutes to run on a 4-core machine. Using the massive parallelism available, gg is able to execute all of these test cases in parallel in less than four minutes, with 99% of the test cases finishing within the first 30 seconds. From a developer's point of view, this improves turnaround time and translates into faster discovery of bugs and regressions.

In addition to software compilation and unit testing, we ported a number of other programs to emit gg IR, including an implementation of ExCamera on gg that, unlike the original implementation, supports memoization and failure recovery, an object recognition task with TensorFlow, and a Fibonacci series program that demonstrates gg abilities on handling *dynamic* execution graphs. For the details of these applications, we refer the reader to our USENIX ATC '19 paper [3].

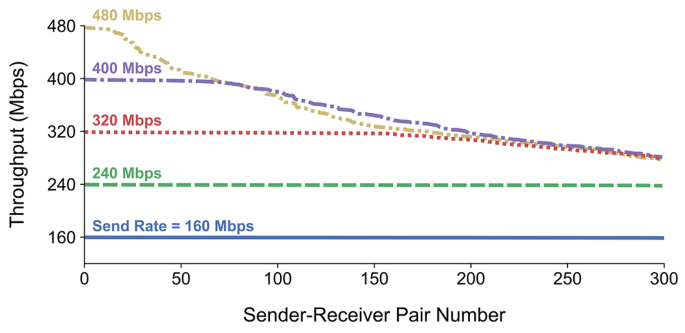


Figure 4: The distribution of achieved network throughputs between pairs of workers at five different send rates on AWS Lambda. Each point corresponds to a sender-receiver pair, and the lines are labeled with their corresponding send rates.

Next Steps: Direct Communication between Workers

Many of the applications that can benefit from *burst-parallel* execution are not *embarrassingly* parallel—they can have complex dataflow graphs and require moving large amounts of data between workers. It has generally been understood that Lambdas cannot accept incoming network connections [5]. As a result, Lambda-computing tools have retreated to exchanging data between workers only indirectly. For example, ExCamera achieves this through TCP connections brokered by a TURN server (each Lambda worker makes an outgoing connection to the server), while PyWren suggests that nodes write to and read from S3, a network blob store. Some of us have developed storage systems like Pocket [7] for ephemeral data storage between workers. However, the latency and throughput limitations introduced by indirect communication (and by mediating inter-node communications through a network file system) are a disqualifier for many applications.

Our preliminary results suggest a more hopeful story for the ability of swarms of cloud functions to tackle communication-heavy workloads, even on current platforms. We have found that on AWS Lambda, workers *can* establish direct connections between one another, and have been able to communicate at up to 600 Mbps using standard NAT-traversal techniques. Figure 4 shows a distribution of achieved network throughputs at five different send rates. For each send rate, we started 600 workers divided into 300 sender-receiver pairs, and each sender transmits UDP datagrams to its pair at that rate for 30 seconds. To be sure, these results indicate variable and unpredictable network performance, but we believe that by designing appropriate protocols and abstractions and failover strategies, direct worker communication can enable a myriad of HPC applications on top of cloud-function platforms.

Our main motivation for this investigation is to build a 3D ray-tracing engine on gg, with the goal of rendering complex scenes with low latency. Currently, the artists who work on 3D scenes rely on high-end machines to iterate on their work—scenes that require tens or sometimes hundreds of gigabytes of memory and take hours to render. Often, the artists must limit the complexity of these scenes (geometry and texture data) by the amount of RAM it is feasible to put in one workstation. For rendering the same scene on RAM-constrained cloud functions, the scene data has to be spread over the workers, which in turn requires low-latency, high-throughput communication between workers to achieve the desired performance. Only further work will tell whether this application can successfully be parallelized to thousands of parallel cloud functions.

Conclusion

We have described gg, a framework that helps developers build and execute burst-parallel applications. gg presents a lightweight, portable abstraction: an intermediate representation (IR) that captures the future execution of a job as a composition of lightweight containers. This lets gg support new and existing applications in various languages that are abstracted from the compute and storage platform and from runtime features that address underlying challenges: dependency management, straggler mitigation, placement, and memoization.

We suspect that cloud functions, as a computing substrate, are in a similar position to that of graphics processing units in the 2000s. At the time, GPUs were designed solely for 3D graphics, but the community gradually recognized that they had become programmable enough to execute some parallel algorithms unrelated to graphics. Over time, this "general-purpose GPU" (GPGPU) movement created systems-support technologies and became a major use of GPUs, especially for physical simulations and deep neural networks.

Cloud functions may tell a similar story. Although intended for asynchronous microservices, we believe that with sufficient effort by the community, the same infrastructure is capable of broad and exciting new applications. Just as GPGPU computing did a decade ago, nontraditional "serverless" computing may have far-reaching effects.

For more information on this project, including our research paper, the code, and quick-start guides, please visit the gg website at https://snr.stanford.edu/gg.

Acknowledgments

We thank the USENIX ATC reviewers and our shepherd, Ed Nightingale, for their helpful comments and suggestions. We are grateful to Geoffrey Voelker, George Porter, Anirudh Sivaraman, Zakir Durumeric, Riad S. Wahby, Liz Izhikevich, and Deepti Raghavan for comments on versions of our research paper. This work was supported by NSF grant CNS-1528197, DARPA grant HR0011-15-20047, and by Google, Huawei, VMware, Dropbox, Facebook, and the Stanford Platform Lab.

References

[1] H. Abelson, G. J. Sussman, with J. Sussman, *Structure and Interpretation of Computer Programs,* 2nd ed. (MIT Press, 1996).

[2] L. Ao, L. Izhikevich, G. M. Voelker, and G. Porter, "Sprocket: A Serverless Video Processing Framework," in *Proceedings of the ACM Symposium on Cloud Computing (SoCC '18),* ACM, pp. 263–274.

[3] S. Fouladi, R. Romero, D. Iter, Q. Li, S. Chatterjee, C. Kozyrakis, M. Zaharia, and K. Winstein, "From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers," in *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC '19),* USENIX Association, 2019.

[4] S. Fouladi, R. S. Wahby, B. Shacklett, K.V. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein, "Encoding, Fast and Slow: Low-Latency Video Processing Using Thousands of Tiny Threads," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI '17),* USENIX Association, 2017, pp. 363–376.

[5] J. M. Hellerstein, J. Faleiro, J. Gonzalez, J. Schleier-Smith, V. Sreekanti, A. Tumanov, and C. Wu, "Serverless Computing: One Step Forward, Two Steps Back," in *CIDR 2019, 9th Biennial Conference on Innovative Data Systems Research,* 2019.

[6] E. Jonas, S. Venkataraman, I. Stoica, and B. Recht, "Occupy the Cloud: Distributed Computing for the 99%," in *Proceedings of the 8th Symposium on Cloud Computing (SoCC 2017).*

[7] A. Klimovic, Y. Wang, P. Stuedi, A. Trivedi, J. Pfefferle, and C. Kozyrakis, "Pocket: Elastic Ephemeral Storage for Serverless Analytics," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI '18)* USENIX Association, 2018, pp. 427–444.

[8] D. Takahashi, "How Pixar Made Monsters University, Its Latest Technological Marvel," VentureBeat, December 2018: https://venturebeat.com/2013/04/24/the-making-of-pixars-latest-technological-marvel-monsters-university/.