FAASTLOOP: Optimizing Loop-Based Applications for Serverless Computing

Shruti Mohanty
The Pennsylvania State University
sxm1743@psu.edu

Vivek M. Bhasi
The Pennsylvania State University
vmb5204@psu.edu

Myungjun Son
The Pennsylvania State University
mjs8014@psu.edu

Mahmut Taylan Kandemir
The Pennsylvania State University
mtk2@psu.edu

Chita Das
The Pennsylvania State University
cxd12@psu.edu

ABSTRACT

Serverless Computing has garnered significant interest for executing High-Performance Computing (HPC) applications in recent years, attracting attention for its elastic scalability, reduced entry barriers, and pay-per-use pricing model. Specifically, highly parallel HPC apps can be divided and offloaded to multiple Serverless Functions (SFs) that execute their respective tasks concurrently and, finally, their results are stored/aggregated. While state-of-the-art user-side serverless frameworks have attempted to fine-tune task division amongst the SFs to optimize for performance and/or cost, they have either used static task division parameters or have only focused on minimizing the number of SFs through task packing. However, these methods treat the HPC code as a black-box and usually require significant manual intervention to find the optimal task division. Since a significant portion of the HPC applications have a loop structure, in this work, we try to answer the following two questions: (i) Can modifying the loop structure in the HPC code, originally optimized for monolithic (non-serverless) frameworks, enhance performance and reduce costs in a serverless architecture?, and (ii) Can we develop a framework that allows for an efficient transition of monolithic code to serverless, with minimum user input?

To this end, we propose a novel framework, FAASTLOOP, which intelligently employs loop-based optimizations (as well as task packing) in SF containers to optimally execute HPC apps across SFs. FAASTLOOP chooses the relevant optimization parameters using statistical models (constructed via app profiling) that are able to predict the relevant performance/cost metrics as a function of our choice of parameters. Our extensive experimental evaluation of FAASTLOOP on the AWS Lambda platform reveals that our framework outperforms state-of-the-art works by up to 3.3× and 2.1×, in terms of end-to-end execution latency and cost, respectively.

SoCC '24, November 20–22, 2024, Redmond, WA, USA

ACM ISBN 979-8-4007-1286-9/24/11.
https://doi.org/10.1145/3698038.3698560

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

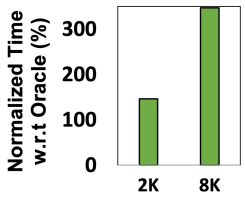
HPC, serverless, compiler, loop optimizations

ACM Reference Format:
Shruti Mohanty, Vivek M. Bhasi, Myungjun Son, Mahmut Taylan Kandemir, and Chita Das. 2024. FAASTLOOP: Optimizing Loop-Based Applications for Serverless Computing. In *ACM Symposium on Cloud Computing (SoCC '24), November 20–22, 2024, Redmond, WA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3698038.3698560

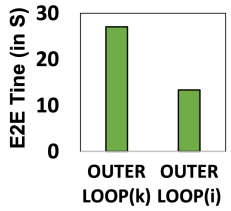
1 INTRODUCTION

A growing number of organizations are turning to cloud platforms for deploying High-Performance Computing (HPC) applications [10, 13, 27, 30, 43, 48], driven by the need for continuously-updated and -managed hardware, software, and tools provided by cloud vendors [1, 4, 25]. Among the public cloud services, serverless computing has seen substantial growth [3, 5, 45], due to its scalability and pay-as-you-go model. In recent years, increasing number of works have used Serverless Functions (SFs) in various HPC applications, including data-parallel scientific computation, machine learning (ML), and parallel video processing [8, 11–14, 22, 28, 31, 34, 36, 38]. This surge in adoption can be attributed to several compelling advantages that SFs offer to highly parallel HPC applications: 1) *Burst-parallel compute on demand*: SF's ability to scale computing resources elastically and quickly, which is ideal for many HPC apps, which

can experience varying levels of parallelism, depending on the size of the input; 2) *Fine-grained billing*: SF's pay-per-use billing model is particularly advantageous for HPC applications with sporadic requests of high-resource demands; and 3) *Simplified management*: SF lowers the barriers to entry imposed by the high costs of maintaining and upgrading clusters.



(a) Normalized execution time with respect to (w.r.t) Oracle (optimal run from offline parameter sweep) comparison for different input matrix sizes (N) with same parallelization strategy. A strategy reasonable for N=2K (0.5× over Oracle), is significantly worse for N=8K (3.3× over Oracle).



(b) End-to-end time comparison of VM optimized parallelization strategy (OUTERLOOP(k)) with another parallelization strategy (OUTERLOOP(i)). VM optimized parallelization strategy (OUTERLOOP(k)) is at least 2× worse in SF architecture.

Figure 1: Exploring the complexities of parallelization strategies in serverless architectures, using the example of GEMM.

But deploying HPC applications on SFs introduces unique challenges distinct from traditional monolithic or distributed setups: 1) *Resource Constraints*: SFs are subject to limitations in resources, e.g. maximum execution time per SF, making it challenging to divide the parallel tasks of HPC app among them, since the work per SF should be such that it can be completed within the time limit etc.; 2) *Location-Agnostic Statelessness*: Unlike classical algorithms that utilize peer-to-peer communication and capitalize on the locality of data and computation, each SF requires individual data transfers to and from cloud storage [35], making code optimized for monolithic architectures often run inefficiently on SFs due to increased data exchanges; and 3) *Scaling Time*: While higher concurrency can yield more 'workers' (function instances) for the app, the time between the start of the first function instance and that of the last instance (referred to as Scaling-Time(ST)) can be prohibitively large and becomes worse with higher concurrency [9, 37]. With multiple such factors affecting performance and cost in varied ways, it becomes challenging to optimize for these metrics.

Previous works to adapt HPC for SFs have primarily centered on integrating SFs within parallel workflow frameworks to address their time constraints [6, 19, 44, 56, 66]. Some have attempted to compress more tasks into a single SF to mitigate scaling times [9, 37], and a few have focused on optimizing task division by manually adjusting and fixing the "block sizes" (work per SF) [47]. However, these approaches often require significant manual effort to find the optimal work division (or "*parallelization strategies*") among SFs (as they are *workload-agnostic*), and, to the best of our knowledge, no previous study has tried to directly – and automatically – modify the HPC code itself to effectively address the above SF challenges.

Algorithm 1 A parallel blocked GEMM [21], where BLK refers to the block size and A_{ik}, B_{kj}, and C_{ij}, of size (BLK,BLK), are blocks of (N,N) *A*, *B*, *andC* matrices, respectively.

```
for k ← 0 to ⌊N/BLK⌋ do parallel        ▷ OUTER-LOOP
    for j ← 0 to ⌊N/BLK⌋ do             ▷ MID-LOOP
        for i ← 0 to ⌊N/BLK⌋ do         ▷ INNER-LOOP
            C_ijk = A_ik · B_kj         ▷ KERNEL
C = Σ_{i,j,k} C_{i,j,k}                 ▷ REDUCE
```

To illustrate that modifying the HPC code carefully can improve performance, we parallelized a blocked matrix multiplication (GEMM), a widely-used kernel in HPC, as depicted in Algorithm 1. We opted for OUTER-LOOP(k) as our default parallelization strategy (i.e., iterations of loop k are distributed across parallel workers) for GEMM on a VM, a widely preferred approach [21, 33]. This approach is quite popular due to its effectiveness in optimizing both temporal and spatial locality in traditional VM settings.

In Figure 1a, we evaluated the impact of using a block size (*BLK*) optimized for one input matrix size (*N*), i.e., 2K, on a different *N*, i.e., 8K. As illustrated, the performance of *N* = 8K is more than 3× that of Oracle[1]. This is because varying *N* could lead to different bottlenecks in the SF framework (details in Section 3). For instance, it directly influences the number of parallel workers (*N*/*BLK*), affecting the scaling time of the SFs [9, 37, 47]. Therefore, *users need to manually adjust block sizes and parallelization strategies to optimize performance across different input sizes, even within a single application.* It is also notable that *BLK*=2K is 0.5× higher than Oracle, indicating potential for further enhancements.

[1]Oracle represents an "optimal run" for a given input size, determined by manually exploring all potential combinations of parameters.

As shown in Figure 1b, with identical *BLK* and *N* values, the VM-optimized loop order (OUTERLOOP(k)) does not give the best results; OUTER-LOOP(i), which is another loop order, outperforms it by 2×. This occurs due to the additional "data movement" in OUTERLOOP(k), a consequence of the "stateless" nature of SFs (details in Section 3). *Therefore, modifying the HPC code for a serverless architecture, even for a single application with varying inputs, is non-trivial.* That is, the performance of a serverless version of a kernel computation depends strongly on how it is parallelized. While these experiments focus on end-to-end time, similar patterns also manifest with regards to cost (i.e., how much is the cost of running a serverless application in cloud).

As mentioned in [37], the resources allocated to SFs tend to be underutilized during the data transfers, a frequent issue due to SFs' stateless nature. This situation presents an opportunity to try and pack more parallel "tasks" of a "job" within a single SF container, which reduces the number of SFs invoked in parallel. Such "task packing" within an SF can decrease the number of SFs needed to run in parallel, directly reducing *both* the scaling time and cost associated with SFs, as also discussed in [9, 37]. However, "over-packing" can cause potential performance degradation due to interference between the tasks, impacting the overall time/cost. Thus, it is important to prudently *trade off* performance degradation with overall time/cost benefits while packing.

Problem. To efficiently parallelize a given HPC job (e.g., a multi-dimensional loop nest) in an SF architecture a user needs to: (i) decide how to restructure the code after understanding the target SF architecture; (ii) how to effectively determine the optimal parameters for parallelizing the code to handle varying inputs; and (iii) how many tasks to pack in a single SF. For an HPC user, to do this manually for each job and each input to the job would be quite tedious and require expert knowledge of the SF framework.

Key Idea. To achieve high performance and/or cost savings (as specified by the user), an *automated framework* that can restructure a monolithic HPC code for an SF deployment, for varying inputs, is needed. We start by observing that, in many popular HPC kernels [9, 16, 41, 47] (e.g., GEMM), the computationally-intensive loop-nests are parallelized to accelerate the job execution. Thus, loop optimization strategies, typically applied by compilers to improve loop executions [24, 49, 54], can be used here to explore different code structures for an HPC job in an SF framework. Furthermore, even greater performance and cost benefits can be achieved by efficiently packing tasks, which reduces the number of SFs spawned. On the other hand, to pick up the right code structure, parameters, and task packing per SF, we need a model capable of predicting the latency components and the overall cost of an SF implementation of an HPC app as a function of the loop parallelization parameters. Finally, the task packing degree can guide the selection of an appropriate "parallelization strategy" and task packing. Note however that, since the tradeoffs in a serverless environment regarding which loop(s) should be parallelized and which tasks could be packed together are in general different from those in a monolithic environment (as both parallelization/packing costs and cloud cost are different in the serverless case), one cannot simply adopt and use the strategies developed in the monolithic context.

Contributions:

- First, we characterize the impact of loop optimizations (different parallelization strategies) and task packing on the end-to-end time and cost of an HPC app on SFs (Section 3).
- Second, based on this characterization, we build an "analytical model" designed to rank the different latency and cost of an SF deployment for a given set of parallelization strategies and packing parameters (Section 4).
- Third, we design FAASTLOOP, the first framework, to the best of our knowledge, to deploy HPC workloads on Serverless Functions (SFs) using loop-based optimization techniques to optimize performance and/or cost (user-specified) (while using additional optimizations like task packing) (Section 5).
- Finally, we evaluate our framework with multiple HPC workloads (Section 7) on AWS Lambda. This evaluation reveals that our proposed FAASTLOOP outperforms the alternate schemes tested by up to 3.3× and 2.1× when optimizing for end-to-end latency and cost, respectively.

2 BACKGROUND

2.1 Serverless Architecture

Serverless Functions (SF), or Function-as-a-Service (FaaS), revolutionized cloud computing by offering unparalleled scalability, parallelism, and a pay-per-use billing model that optimizes costs. However, serverless functions are "stateless", necessitating data transfers, and have resource limitations, with respect to concurrency, execution time and memory, imposed by serverless platforms. Additionally, adapting monolithic applications to serverless-ready code can be very challenging as the users need to understand SF-specific coding practices.

We depict the workflow of a serverless application using Figure 2. Here, "VM" refers to the user's computing environment, which can be a cloud-hosted VM or a personal laptop. ❶ Parallel user requests to SFs are triggered by API calls or events. ❷ The API gateway *auto-scales* SF containers and schedules the incoming requests onto them. ❸ Once the SF starts, it downloads the data needed for subsequent computation. ❹ The SF then performs its computation on the data (we call this "kernel execution"). ❺ If required, the result of the computation is uploaded into storage. ❻ Additionally, if needed, the results of all the SFs are downloaded

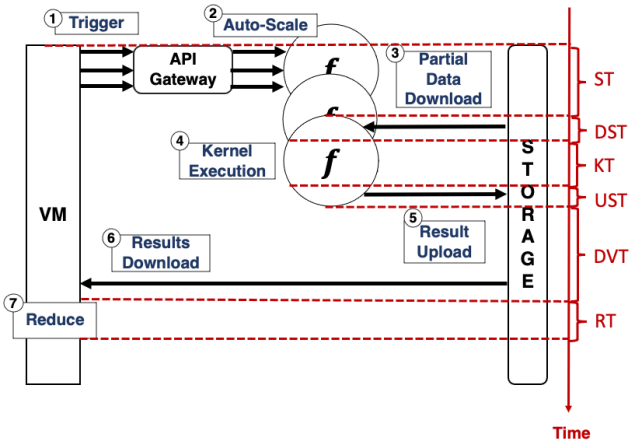


Figure 2: Serverless workflow and associated latency terms.

by the user(s), after which ❼ additional post-processing can be applied. Note that for the apps we consider, this post-processing is in the form of a 'many-to-one' reduce step. The associated latency for each step is shown in Table 1.

From the latency components shown in Table 1, the total end-to-end latency for this workflow, T_{e2e}, is given by:

$$T_{e2e} = ST + (DST + KT + UST) + DVT + RT. \quad (1)$$

ST denotes the time spent by the serverless platform to scale containers to serve the parallel requests. DST, KT, and UST collectively represent the average runtime of a single SF (also denoted as SFT). And, DVT and RT are time spent in user's VM to assimilate the results. Based on these, the cost equation for the workflow would be:

$$Cost_{e2e} = (M_{SF} \cdot CC_{SF} \cdot (DST + KT + UST) + RC_{SF}) \cdot N_{Req}, \quad (2)$$

where M_{SF} is the memory of an SF, CC_{SF} and RC_{SF} are, respectively, the compute and request constants of the SF framework (constants used by SF platforms to calibrate cost) [2] and N_{Req} is the number of SF requests. While these equations are representative of AWS's framework, there are only minor variations in the equations across different serverless platforms [23, 39]. As can be observed from Equations 1 and 2, longer end-to-end execution time may *not* necessarily result in higher costs. This is because end-to-end time could increase because of higher scaling time (ST) or downloading results to VM time (DVT), but this may not affect cost, as cost is only dependent on SF runtime (SFT) and N_{Req}. *Therefore, designing an SF application for different objectives, such as minimizing cost or latency, can lead to significantly different code structures.*

Table 1: Serverless latency terms and descriptions.

Time	Description
ST	Time from request initiation to all parallel requests running on SF
DST	Average data download time for an SF
KT	Average kernel execution time by an SF
UST	Average data upload time for an SF
DVT	Data download time to a VM
RT	Time to aggregate various results

2.2 Related Works

Various studies have explored integrating serverless architectures with HPC applications to leverage the scalability, cost-efficiency and ease of management of SFs.

A significant portion of the literature has focused on using SFs within a parallel workflow management framework to manage the parallel threads of HPC applications [6, 19, 44, 56, 66]. These efforts often focus on reducing cost, developing fault-tolerant systems or managing the inherent limitations of SFs (such as caps on execution time, maximum resource allocations, etc.). [42, 66] introduce checkpointing mechanisms to maintain state across SF executions, ensuring that long-running HPC tasks can resume after hitting execution time limits. In comparison, [64] has optimized scheduling the threads on serverless platforms to be sensitive to both performance and cost. Some works try to find the ideal SF size [17, 65] that balances performance and cost. However, they overlook other serverless architecture optimizations for performance, such as scaling issues and statelessness, and use the code *as-is* without modifications for serverless environments.

A few recent works [9, 37, 47], have attempted to address some of the above challenges. More specifically, [47] manages data partitioning and parallel SF execution through loop blocking/tiling, adhering to the resource limits of serverless environments. However, it sets a static block size for each application, which either remains unchanged for different inputs or requires manual recalibration by an expert for each new input. This fixed block size approach is inefficient – too large a block size underutilizes potential parallelism for smaller inputs, while too small a block size increases scaling time for larger inputs due to excessive concurrency. On the other hand, other studies, like [9, 37], focus on reducing scaling time and adhering to the concurrency limits in SFs by *packing* more tasks into each SF container. This strategy optimizes task interference and addresses the underutilization of computational resources during data loading and unloading, a common issue due to the stateless nature of SFs. However, such studies do *not* determine the optimal block size or modify the HPC code to fit the serverless architecture. Without these adjustments, task packing alone may not effectively reduce time and costs (as we will demonstrate in Section 7).

While the above works offer valuable strategies for optimizing serverless architecture knobs for parallel HPC tasks,

they overlook the possibility of adapting the HPC application code for serverless environments. Additionally, these approaches often require significant manual effort and/or learning a new domain-specific language to achieve efficient runtime. In contrast, our work focuses on automatically restructuring code by recognizing that loop structures are a fundamental aspect of many HPC apps and, in turn, leveraging existing research on loop optimizations [7, 24, 40, 49, 54, 61]. We explore the effectiveness of the loop optimization techniques for a serverless architecture and try to answer the following question: **can loop optimizations help optimize performance and cost for a SF deployment?** We *automate* the restructuring process for SFs by utilizing a compiler that automatically converts monolithic code into FaaS-ready code. Additionally, we also automate SF task packing so as to spawn the optimal number of SFs that also efficiently trades off scaling time and task interference.

3 MOTIVATIONAL ANALYSIS

One can observe from Section 2 that the manner in which task partitioning/division (parallelization) of an HPC application/job between SFs is done is critical to serverless performance and cost due to its influence on the scaling delay, data movement, and kernel execution time (from Equation 1). On top of that, task division must also comply with the resource constraints imposed by SF platforms. Previous works have typically relied on a fixed code structure per task. In contrast, in this work, we examine optimization strategies that modify code structure and enhance resource utilization of SFs, assessing their impact on the overall execution time and cost of HPC workloads.

3.1 Loop Optimizations for Serverless Task Sizing

It is well-known that many HPC applications/kernels have loop nests which can be parallelized [15]. While loop optimization techniques have been studied extensively in the realm of HPC [7, 24, 29, 40, 49, 54, 61], they have yet to be applied within the context of serverless task sizing. Motivated by this observation, this section explores the potential of finding the optimal task division (partitioning) using three specific loop optimization strategies, namely, *Loop Blocking/Tiling*, *Loop Interchanging*, and *Multi-Level Parallelism*. To this end, we conduct experiments using the Matrix-Multiplication (GEMM) algorithm, as it encapsulates common characteristics and computational patterns found in a broad range of HPC algorithms [57]. Algorithm 1 illustrates a simplified version of a Parallel Blocked GEMM algorithm [21, 50] that will be used. While we used GEMM as an example, these methods are applicable to any nested loop-based applications without inter-loop dependencies. For instance, consider a workload doing pairwise human protein comparisons [41]. This process uses two nested loops, with the inner kernel executing the Smith-Waterman algorithm, significantly differing from GEMM operations. As shown in section 7, we have assessed FAASTLOOP's performance on Smith-Waterman and other loop-based applications, demonstrating its applicability beyond GEMM to a wide range of loop-based scenarios.

Algorithm 2 OUTERLOOP Parallelization, where the OUTERLOOP of GEMM from Algorithm 1 runs in parallel using SFs.

In SF:
function SF_WORKER(i)
  Download_Partition(A_i, B)
  for $j \leftarrow 0$ to $\lfloor \frac{N}{BLK} \rfloor$ **do** ▷ *MID-LOOP*
    for $k \leftarrow 0$ to $\lfloor \frac{N}{BLK} \rfloor$ **do** ▷ *INNER-LOOP*
      $C_i += A_{ik} \cdot B_{kj}$ ▷ *KERNEL*
  Upload_Partition(C_i)

In VM:
for $i \leftarrow 0$ to $\lfloor \frac{N}{BLK} \rfloor$ **do parallel** ▷ *OUTER-LOOP*
  SF_WORKER(i)
Download_All_Partitions()
// ***Compute C via a sum reduction***
$C = \sum_i C_i$

3.1.1 ***LOOP BLOCKING.*** Loop blocking (also known as "tiling") is a widely-used technique for granularity control of parallelism and data movement/reuse optimization for HPC workloads [26, 52, 59, 62]. It segments the problem (set of computations within a loop nest) into smaller blocks which can, in turn, be distributed among different parallel processing units. Further, by using sufficiently small (iteration) blocks, data can be made to fit in the processor's cache which helps decrease the frequency and volume of data movements.

While state-of-the-art serverless works [9, 47] attempt to perform task division of HPC jobs across SFs (similar to blocking), they use a fixed block size for an application (which may have been chosen by an expert-user for a particular input size for that application). Below, we demonstrate, how using a fixed block size (*BLK*) across all inputs for an application (GEMM, in our example) can lead to sub-optimal task division, thereby, affecting performance as well as cost.

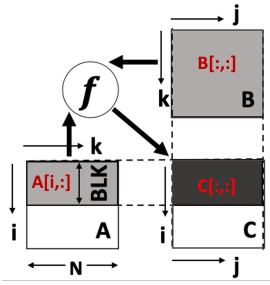


Figure 3: Loop Blocking: Impact of Blocksize (*BLK*) on Data Division in OUTER-LOOP Parallelized GEMM for an Input Size (N).

As seen in Algorithm 2, we run the OUTER-LOOP (i-loop) in parallel. Here, the 'i-loop'

corresponds to running $\frac{N}{BLK}$ parallel SFs, where N is the input size and BLK is the block size. Each SF worker downloads the necessary data ($A_{i,k}$, $B_{k,j}$) for computing their portion of the matrix multiplication using the loops k and j (loop variables within the SF). Following the computation, the SF worker uploads the result to storage. Figure 3 shows the data partition for a single SF for Algorithm 2. As matrix A is indexed by (i, k) (as shown in the kernel of Algorithm 2), SF needs to download A[i,:] which is of size $BLK \times N$. Similarly, as matrix B is indexed by (k, j), SF needs to download B[:,:], which is of size $N \times N$ (Figure 3). Finally, the result matrix C, which is of size $BLK \times N$, is uploaded. Building upon this, we define five SF **performance metrics** that would help us in analyzing how BLK affects latency and cost.

$$Ops_{\text{perSF}} = RangeOf(jloop) \cdot RangeOf(kloop) \cdot Ops(innerKernel) = \frac{N}{BLK} \cdot \frac{N}{BLK} \cdot BLK^3 = N \cdot N \cdot BLK \quad (3)$$

$$DD_{\text{perSF}} = Sizeof(A[i,:]) + Sizeof(B[:,:]) = N \cdot BLK + N \cdot N \quad (4)$$

$$DU_{\text{perSF}} = \text{Sizeof}(C[i,:]) = N \cdot BLK \quad (5)$$

$$numSF = Rangeof(OUTERLOOP) = \frac{N}{BLK} \quad (6)$$

$$DD_{\text{VM}} = numSF \cdot Sizeof(C[i,:]) = N \cdot N \quad (7)$$

Here, Ops_{perSF}, DD_{perSF}, and DU_{perSF} refer to operations per SF, data downloaded per SF, and data uploaded per SF, respectively. $numSF$ is the total number of parallel SFs formed and DD_{VM} is the data downloaded by the VM. Ops_{perSF} can be determined by multiplying *RangeOf(jloop)*, *RangeOf(kloop)* (i.e., the loops inside an SF, from Algorithm 2), and *Ops (innerKernel)*. Since the innerKernel is a GEMM (matrix multiply) operation, the number of operations (*Ops(inner Kernel)*) is BLK^3. DD_{perSF} is dependent on the partitioning of matrices A and B, downloaded by an SF. As shown in Figure 3, the partition size of matrix A equals N×BLK and matrix B equals N×N. Similarly, DU_{perSF} depends on the partition size of matrix C equals N×BLK. $numSF$ denotes the count of parallel SFs, which is determined by the range of the outer loop (i loop) in Algorithm 2, calculated as $\frac{N}{BLK}$. DD_{VM} refers to the data downloaded by the VM which comprises the results produced by all parallel SFs. It is determined by multiplying the number of parallel SFs (i.e., $numSF$) and each result ($Sizeof(C[i,:])$ = N×BLK).

From the above equations, we observe that BLK can affect performance/cost in various ways. Firstly, increasing BLK can effectively reduce the number of parallel SFs ($numSF$), which in turn, helps in lowering the *scaling time*. Secondly, decreasing BLK can reduce the data download/upload (DD_{perSF}/ DU_{perSF}), thereby reducing the *data transfer time*. Thirdly, given the *resource constraints* imposed by an SF provider,

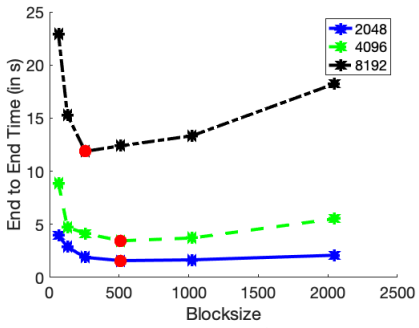


Figure 4: Impact of Blocksize (BLK) on end-to-end latency for different Input Sizes (N, shown in legend) on GEMM. The minima found for each line, depicted by the red dot, are not same for all N.

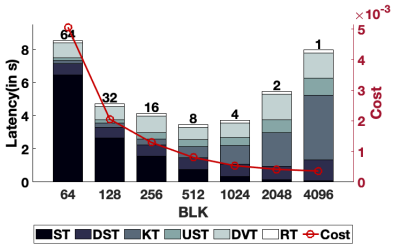


Figure 5: For Input Size N = 4096, splitting the end-to-end latency according to Figure 2. The number on top of the bars represents the total number of parallel SFs.

tuning BLK can help to adhere to those limits on concurrency ($numSF$), memory (DD_{perSF}/ DU_{perSF}) and execution time (Ops_{perSF}). Fourthly, for any change in input size (N), adjusting BLK allows for effective optimization of task sizing in response to the new N, since most terms are a function of N and BLK.

To demonstrate the validity of the above reasoning, we deployed Algorithm 2 on AWS using EC2 (8 vCPUs) and Lambdas. We experimented with varying block sizes (BLK) for different input sizes (N) (Figure 4). The results collected clearly indicate that achieving minimum end-to-end execution time requires different BLK values for different N values. As an example, when N=2K or 4K, a BLK of 512 is optimal for minimizing end-to-end execution latency, whereas, for N=8K, a BLK of 256 performs the best. Similarly, we also observe that, for optimizing cost, different N values prefer different values for the BLK parameter.[2]

We performed a fine-grained analysis to see **why** BLK **affects performance/cost**. For this, we observe the distinct latency components of end-to-end execution time for a fixed input size, N=4k (as shown in Figure 5), for a sweep across various BLKs. For smaller block sizes (BLK), where $numSF$(Equation 6) is higher, the scaling time (ST) becomes

[2]Not shown due to space constraints.

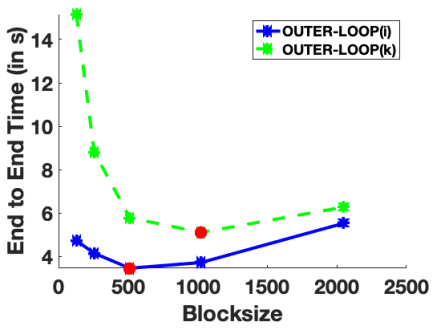


Figure 6: Impact of interchanging loop i and loop k with different block sizes in OUTER-LOOP Parallelized GEMM, for an input size of N=4096. Minima of each line, depicted by the red dot, is different.

Table 2: Comparing OUTER-LOOP(i) and OUTER-LOOP(k).

Metric	OUTER-LOOP(i)	OUTER-LOOP(k)
Ops_{perSF}	$N \cdot N \cdot BLK$	$N \cdot N \cdot BLK$
DD_{perSF}	$N \cdot BLK + N \cdot N$	$N \cdot BLK + N \cdot BLK$
DU_{perSF}	$N \cdot BLK$	$N \cdot N$
$numSF$	$\frac{N}{BLK}$	$\frac{N}{BLK}$
DD_{VM}	$N \cdot BLK \cdot \frac{N}{BLK} = N \cdot N$	$N \cdot N \cdot \frac{N}{BLK}$

the primary bottleneck. Conversely, for larger BLKs, the runtime of SF (comprising DST, KT, and UST) increases as the task size increases. Thus, a user needs to carefully *trade off* these metrics to find an "optimum point" where end-to-end latency is minimized. Note also that the minimum latency is reached with a BLK of 512, yet the most cost-effective solution is achieved with a BLK of 4096, indicating that the selection of BLK depends on the user's specific objective (minimizing latency versus cost).

Takeaway 1: *Adjusting block size is crucial for "balancing" SF runtime and scaling time across different input sizes, and it varies based on whether the goal is to minimize cost or latency.*

In a typical setup for VMs, the Loop Interchange optimization technique is used to rearrange nested loops at the compiler level to improve data locality and efficiency. However, for SFs, this optimization must be adapted to cater to the short-lived and distributed characteristics of serverless computing, diverging from the static and centralized nature of VMs.

3.1.2 ***LOOP INTERCHANGE. How are the five performance metrics affected by loop interchange?*** Table 2 presents a comparison of the five performance metrics (from Eqs. 3, 4, 5, 6, and 7) for a scenario where the i loop variable (OUTER-LOOP(i)) in Algorithm 2 is interchanged with the k loop variable (OUTER-LOOP(k)). We can observe that, even though the total Operations per SF (Ops_{perSF}), total data movement per SF (DD_{perSF} + DU_{perSF}) and number of parallel SF ($numSF$) remain the same for both the algorithms, the total data downloaded by the VM increases by a factor of $\frac{N}{BLK}$ for OUTERLOOP(k). To grasp the shift in data division when moving from OUTERLOOP(i) to OUTERLOOP(k), we compare the data layout of OUTERLOOP(i) (from Figure 3), with that of OUTERLOOP(k) (in Figure 7). It can be observed that the data partition size of matrix A remains the same ($N \cdot BLK$), but the partition sizes for matrices B and C have been interchanged. For OUTERLOOP(k), the size of matrix C increases from $N \cdot BLK$ to $N \cdot N$. Matrix C, being the result matrix, is downloaded by the VM for each and every parallel SF, which has a significant impact on the overall end-to-end latency.

To validate the above argument, we execute OUTER-LOOP(i) and OUTER-LOOP(k) for N = 4K on AWS. The result across different block sizes is shown in Figure 6. We observe that OUTERLOOP(k) exhibits higher latency versus OUTER-LOOP(i). Furthermore, as mentioned above, the difference in latency between the two loop structures, proportional to $\frac{N}{BLK}$, decreases with larger block sizes (Figure 6).

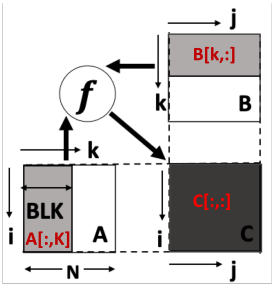


Figure 7: Impact of interchanging loop i and loop k on data division in OUTER-LOOP Parallelized GEMM for input size N.

Here, we make an interesting observation: the ideal block size for achieving the lowest latency may differ based on the specific loop structure used. Note that the cost for both runs remains the same as the runtime of SF and the number of parallel SFs is the same for both the loop structures.

Takeaway 2: *Depending on the loop structure, modifying the loop order may affect a subset of the SF performance metrics, but each change in loop structure requires recalibrating the block size to minimize overall time/cost. Some loop structures will perform worse than others, irrespective of the input size.*

3.1.3 ***MULTI-LEVEL PARALLELISM.*** Multilevel parallelism is a technique where the compiler applies loop coalescing optimization that combines two or more nested loops into a single loop to allow multilevel parallelism. As mentioned in [58], employing coalescing increases the parallelism opportunities as more than one loop can effectively

Table 3: Comparing OUTER-LOOP, MIDDLE-LOOP and INNER-LOOP for LOOP order ijk of GEMM.

Metric	OUTER-LOOP	MID-LOOP	INNER-LOOP
Ops_{perSF}	$N.N.BLK$	$N.BLK.BLK$	$BLK.BLK.BLK$
DD_{perSF}	$N.BLK + N.N$	$2.N.BLK$	$2.BLK.BLK$
DU_{perSF}	$N.BLK$	BLK^2	BLK^2
$numSF$	$\frac{N}{BLK}$	$\frac{N}{BLK}^2$	$\frac{N}{BLK}^3$
DD_{VM}	$\frac{N}{BLK}.N.BLK = N^2$	$\frac{N}{BLK}^2.BLK^2 = N^2$	$\frac{N}{BLK}^3.BLK.BLK = N^2.\frac{N}{BLK}$

run in parallel if there are no dependencies across the loops (i.e., no loop-carried dependency). When we coalesce the middle loop with the outer loop (called MID-LOOP parallel), as shown in Algorithm 3, the parallelism increases by a factor of $\frac{N}{BLK}$ over the OUTER-LOOP parallelization (Algorithm 2). Similarly, when we coalesce all three loops (referred to as INNER-LOOP parallel), the parallelism increases by a factor of $\frac{N}{BLK}^2$ over OUTER-LOOP parallelization.

How are the five performance metrics affected by multilevel parallelism? To understand how each level of parallelism influences the performance and cost, we compare the previous 5 performance metrics in Table 3. We observe that, as each new level of parallelism is introduced, the metrics associated with "per SF" (i.e., Ops_{perSF}, DD_{perSF} and DU_{perSF}) become more sensitive to change in block size (BLK). Since BLK is less than N, the metrics associated with "per SF" generally decrease with additional parallelism. However, the number of parallel SFs ($numSF$) shows an exponential increase. The total amount of data downloaded by the VM (DD_{VM}) depends on (i) the data uploaded by the SF (DU_{perSF}), which decreases with more parallelism, and (ii) the number of parallel SF ($numSF$), which, in contrast, increases with each additional level of parallelism. At first, these increasing and decreasing trends neutralize each other; for instance, the initial increase in parallelism levels (from OUTER-LOOP to MID-LOOP) results in no change in DD_{VM}. However, in the case of INNER-LOOP, the exponential increase in $numSF$ notably elevates the DD_{VM}, scaling it by a factor of $\frac{N}{BLK}$. Similarly, Cost, which depends on the runtime of SF ($Ops_{perSF} + DD_{perSF} + DU_{perSF}$) and the number of parallel SF ($numSF$), also follows a similar trend as DD_{VM}.

Given the above ways in which multi-level parallelism influences the various performance metrics of serverless execution, it can, therefore, help navigate the complexities of the SF infrastructure. It can (i) regulate the number of parallel SFs ($numSF$), which is closely tied to the *scaling time*; (ii) impact *data transfer times* by modulating data uploads and downloads; and, (iii) ensure compliance with SF resource constraints (through our choice of parallelism level).

Algorithm 3 MIDLOOP Parallel GEMM

In SF:
function SF_WORKER(i, j)
  Download_Partition(A_i, B_j)
  for $k \leftarrow 0$ to $\lfloor \frac{N}{BLK} \rfloor$ **do**
    $C_{ij} += A_{ik} \cdot B_{kj}$
  Upload_Partition(C_{ij})

In VM:
for $i, j \leftarrow 0$ to $\lfloor \frac{N}{BLK} \rfloor$ **do parallel** ▷ *MID-LOOP*
  SF_WORKER(i, j)
Download_All_Partitions()
// ***Compute C via a sum reduction***
$C = \sum_{i,j} C_{i,j}$

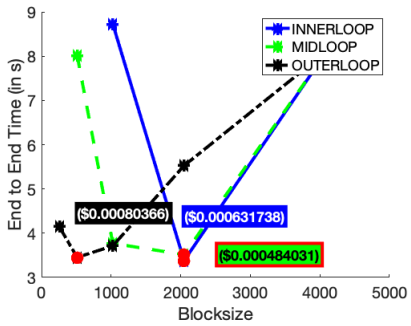


Figure 8: Impact of Multilevel Parallelism with different Blocksizes in GEMM on End-to-End Latency for input size, N = 4096. Minima of each line is depicted by the red dot. Cost (in $) for each minima is written.

To demonstrate the above, we executed OUTER-LOOP, MID-LOOP, and INNER-LOOP, for N = 4K on AWS. The results, across different block sizes, are plotted in Figure 8. We can see that for smaller block sizes (BLK), OUTER-LOOP outperforms MID and INNER-LOOP. This is primarily because the number of parallel SFs ($numSF$), which is dependent on the ratio $\frac{N}{BLK}$, increases exponentially with more parallelism. However, as BLK increases, a balance is struck between the decreasing $\frac{N}{BLK}$ ratio and the rising "per SF" performance metrics for MID and INNER-LOOP. This minimizes end-to-end latency, with larger BLKs reducing the number of SFs and enhancing per SF efficiency, for better overall performance. Here, we display the cost for minimum end-to-end latency for each level of parallelism. Even though all parallelism levels show similar minimum latency, the MID-LOOP stands out as being nearly 40% more cost-effective. In the OUTER-LOOP, $numSFs$ is low, but runtime increases for each SF. Conversely, in the INNER-LOOP, there is a cubic increase in $numSF$ even though the runtime per SF might be shorter. The MID-LOOP strikes an optimal balance between these two factors, leading to its higher cost efficiency.

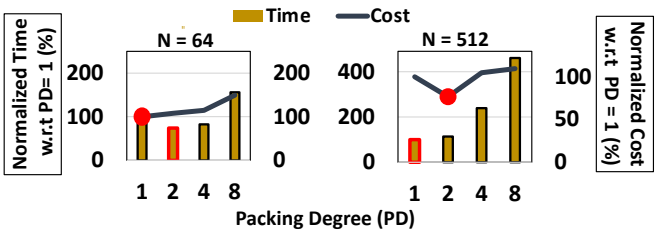


Figure 9: For Input Size N = 4096, impact of task packing on different block sizes (BLK =64 and 512) for GEMM on End-to-End Latency and Cost. Min cost is depicted by the red dot.

Takeaway 3: *Parallelizing different loops offers more precise task sizing control beyond just adjusting block size, enabling significant cost reductions even with similar latencies.*

Note that we have focused on HPC applications with perfectly-nested loops devoid of inter-loop dependencies (since this is representative of many HPC workloads [20, 41, 47]). We limit the scope of our work to the loop-based techniques suitable for such applications and will explore other methods, such as loop fission and fusion (which may address more intricate loop structures), as part of our future work.

3.2 In-SF Optimization: Task Packing

So far, our focus has been on loop optimizations for addressing the challenges inherent in adapting HPC code to the serverless framework, specifically targeting issues like scaling time, data transfer time, and resource constraints. However, beyond these loop-level adjustments, we can also turn to system-level optimizations, such as "task packing", as SFs are underutilized during the data transfer phases. Specifically, by packing more parallel tasks of a job within a single SF container, we can effectively reduce the number of parallel SFs (*numSF*). This directly affects *scaling time* and aids in managing resource constraints, such as concurrency. Through task packing, more tasks can fit into a single SF, which can decrease *numSF*, potentially improving both latency and cost. However, if an SF container is over-packed, the tasks can interfere with each other, which can negatively impact the SF runtime, leading to performance and cost degradation. Therefore, it is crucial to find a "balance" in task packing that minimizes scaling time without reaching a point where the task interference becomes counterproductive.

To demonstrate the effect of task packing on latency and cost, we executed GEMM OUTERLOOP(i) of input size N = 4K on AWS. The results for two block sizes (BLK=64, 512) are shown in Figure 9. The minimum time and cost for each BLK are depicted by a red bar and circle, respectively. For BLK = 64, although packing does not reduce costs, it improves overall time due to a significant reduction in scaling time, which accounts for over 80% of the total time as shown in Figure 5. The benefits of reduced scaling time at Packing Degree (PD)=2 offset the increased runtime from task interference. However, for PD greater than 2, the gains from reduced scaling time diminish and overall time increases due to escalating task interference. Conversely, the cost at PD=2 does not benefit similarly; the savings from halving the number of SFs are negated by the longer SF runtime caused by task interference. In the case of BLK = 512, the dynamics change as the scaling time is more effectively balanced with SF runtime. Increases in SF runtime directly extend the overall runtime since reductions in scaling time are minimal. However, for cost considerations, reducing the number of SFs at PD=2 proves economical despite increased task interference, as the cost savings from fewer SFs outweigh the downsides. Therefore, the effectiveness of packing in reducing cost or latency depends crucially on the specific bottlenecks encountered.

Takeaway 4: *The impact of task packing on cost and latency varies significantly based on the parallelization strategy employed and its system bottlenecks.*

As established, finding the right task size for parallelizing an HPC application via SFs is not trivial: a user needs to manually sweep across several block sizes, reorder loops, decide which level to parallelize and choose a packing degree (taking into account SF-specific bottlenecks and cloud costs), to determine the best "parallelization/optimization parameters" to deploy the application. Thus, the size of parameter search space for a single input for an application is the product of (i) *number of block sizes*, (ii) *number of nested parallelizable loops*, (iii) *number of distinct loop orders*, and (iv) *possible packing degrees*. This space is clearly quite large, and motivated by this observation, our proposed FAASTLOOP automates the process of *efficiently exploring this search space* by leveraging profiling to model different latency components of serverless for an application with respect to different inputs. We discuss this in the next section.

4 ANALYTICAL MODELING

As highlighted in the previous section, finding the right parallelization strategy for optimally distributing a job across SFs, whether to minimize time or cost, involves *exploring a vast search space*, even when considering just a single input size for an application. In fact, manually adjusting these parameters for each input per application is impractical due to the significant time and cost it involves. This motivates us to form an analytical model based on meticulously collected offline profiled data that would help in identifying the optimal parameters for executing a loop-based application on SFs, either for reducing cost/latency while staying within

the SF's constraints. We also update our models periodically at runtime.

Recall from Section 2.1 that, the equations for end-to-end latency (T_{e2e}) and cost ($Cost_{e2e}$) are:

- $T_{e2e} = ST + (DST + KT + UST) + DVT + RT$.
- $Cost_{e2e} = (M_{SF} \cdot CC_{SF} \cdot (DST + KT + UST) + RC_{SF}) \cdot numSF$.

Additionally, many SF frameworks (like AWS, Azure, etc.) also impose certain **resource constraints** that these equations also have to comply with, such as:

- $SFT_{max} >= DST + KT + UST$.
- $M_{max} >= M_{SF}$.
- $numSF_{max} >= numSF$.

Here, SFT_{max} represents the maximum allowable runtime per SF, M_{max} denotes the maximum memory allocation for a single SF, and $numSF_{max}$ specifies the maximum SF concurrency permitted per user per region. Thus, to rank the T_{e2e} and $Cost_{e2e}$ of different "parallelization strategies", we model the various latency components of an SF-based HPC app (described in Section 2.1 and Figure 2). Building on the insights from Section 3, the five performance metrics, namely, i) Operations per SF, ii) Data Download per SF, iii) Upload per SF, iv) Number of Parallel SFs, and v) Data Download to VM, enable us to model the various latency components effectively.

4.1 Modelling Individual Latency Components

We model the latency components using profiled data from GEMM experiments on AWS EC2 (8 vCPU) and Lambdas (3400MB). We expect these models to suit other serverless frameworks with minimal modifications (probably with minor changes such as those to constant terms). We gauge model accuracy by employing the adjusted R-squared score (>0.8), which offers valuable insights into variable dependencies.

4.1.1 SCALING TIME (ST). From our experiments conducted with different AWS Lambda memory configurations and request rates, we observe that the Scaling Time (ST) is primarily influenced by the number of parallel SF requests (N_{req}), a point also affirmed in [9]. Thus, we construct a "linear model" of the form $ST = \beta.N_{req} + \alpha$, to capture this relationship, where α and β are SF platform-dependent constants.

4.1.2 SERVERLESS FUNCTION TIME (SFT). Serverless Function Time (SFT) is formed by three components, namely, i) Data Download Time by SF (DST), ii) Kernel Execution Time (KT), and iii) Data Upload Time by SF (UST): $SFT = DST + KT + UST$. Data Download and Upload per SF Time (DST & UST) are directly dependent on the size of the data being downloaded (DD_{perSF}) or uploaded (DU_{perSF}). We observe from our profiling results that the data transfer time (both DST and UST) is linearly dependent on the data size (albeit slight fluctuations), for a fixed SF configuration. Thus, both DST and UST have the same linear model represented by $T_{download} = \beta.DataSize + \alpha$, where $T_{download}$ is either DST or UST, and $DataSize$ is DD_{perSF} and DU_{perSF} corresponding to DST and UST, respectively. On the other hand, we observed Kernel Execution Time (KT) is dependent on the number of operations in each serverless function (Ops_{perSF}). We construct the corresponding model as $KT = \beta.Ops_{perSF} + \alpha$.

4.1.3 DATA DOWNLOAD TIME BY VM (DVT) and REDUCTION TIME (RT). Data downloaded by VM is dependent on the number of parallel SF requests (N_{req}) and the data size uploaded by each SF (DU_{perSF}). The total amount of data that is downloaded is $N_{req}.DU_{perSF}$, but we download it *in parallel*. This reduces the overall download time because we can fetch multiple chunks of data concurrently. To account for this effect, we introduce a parameter γ to calibrate the influence of parallel downloading on the download time. Specifically, we model it as $DVT = \beta.DU_{perSF}.(N_{req} - \gamma) + \alpha$. The reduction time step is a synchronization step where a many-to-one reduction is performed. Thus, we observe that it depends on the total data that is being reduced, modeled as $RT = \beta.DD_{perSF}.(N_{req}) + \alpha$.

4.1.4 ACCOUNTING FOR PACKING DEGREE (PD). In addition to the influence of the loop-based optimizations on the serverless latency components, we must also consider how task packing (mentioned in Section 3) affects them. We take inspiration from the model described in [9] for modeling the influence of packing degree (which is simply the number of tasks packed per SF container) on SFT. Thus, when packing multiple tasks into an SF, SFT is modeled as $SFT_i = \alpha * ((SFT_1 - \beta) * e^{(\gamma * (M_i - \delta))} + \epsilon$, where SFT_1 is SFT for PD of 1 (as shown previously in the calculation of SFT) and M_i is the memory used by the SF for Packing Degree (PD) of i.

Forming the Latency, Cost and Resource Constraint Models: We extract the five performance metrics from the application's profiled data. These metrics are then used to fine-tune the time models, ensuring they accurately reflect the application's performance. The time model equation (T_{e2e}) is formed by aggregating all the individual time models (Equation 1). The cost model equation ($Cost_{e2e}$) is created by integrating specific elements from the time model equations (SFT) with the performance metrics (M_{SF} and N_{req}). The resource constraint equations are also derived from the performance metrics. it is to be noted that these equations are critical in defining the boundaries within which the app can function optimally. We get (T_{e2e}) and ($Cost_{e2e}$) for each profiled combination of parameters (block size, parallelization strategies, and packing degree), for different inputs. Here,

for each input, the top 10 configurations for each user objective (minimizing T_{e2e} and $Cost_{e2e}$) are saved. By having these configurations readily available, the runtime optimizer can quickly choose the most effective parameters for any given input.

5 OVERALL DESIGN OF FAASTLOOP

Putting together all the optimizations discussed thus far, we propose a novel serverless user-side framework, FAASTLOOP, which *automates* optimal task divisioning across SFs, to optimize for a user-specified cost/latency goal. We give an overview of FAASTLOOP's design in Figure 10.

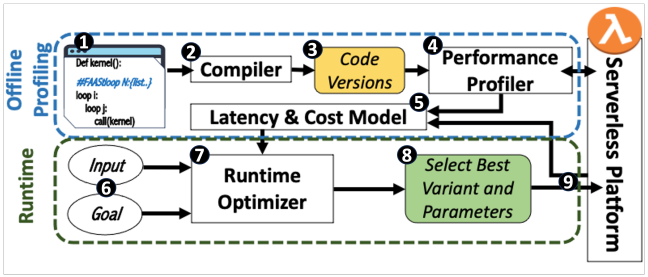


Figure 10: Overview of FAASTLOOP.

End-to-End Workflow: Before deployment, ❶ Users annotate their code for parallelization, preparing it for compilation by the FAASTLOOP compiler. ❷ Compiler then applies loop optimizations to generate various static versions of the code ❸. ❹ Each version is subsequently profiled across different block sizes and packing degrees to assess performance. Based on this profiling, ❺ latency and cost models are developed, which serve to guide optimizations during runtime.

During runtime, ❻ the user inputs the size and specific objectives—such as minimizing cost or latency—of the current run. Using these inputs, ❼ the runtime optimizer examines the parameter space, guided by the previously formed models and static code versions, to identify the most efficient execution plan (❽). Finally, ❾ the optimized code version is deployed on the Serverless Function (SF) according to the user's goals, ensuring that the code runs optimally in a serverless environment. Additionally, during each runtime session, we update our model parameters to adapt to fluctuations from the cloud provider, which tend to be minor over short periods.

Detailed functionalities of the key components of FAASTLOOP framework are summarized below:

User annotations: Users can annotate their code to optimize loop execution, similar to how they might use OpenMP [55] for parallel programming. They can use the *pragma 'faastloop'* to decorate a dependency-free loop, also specifying some expected input sizes (N), as shown in Figure 10. Even though FAASTLOOP can handle new unseen inputs, the expected input size specification helps to restrict the search space during runtime. Since the pragma is applied in a manner akin to OpenMP, HPC users will find it intuitive and beneficial for scaling code without requiring additional training.

Compiler: As part of its offline profiling phase, FAASTLOOP employs a compiler that processes user-submitted HPC app code, consisting of blocked loops annotated with user-provided pragmas that specify the part of the code that is to be executed using SFs. It generates various "code versions" (akin to the methods described in [51]) from the user code, each implementing a combination of loop optimizations as detailed in Section 3, that are ready for FaaS deployment. These code versions (which also have lightweight profiling "hooks") will then be used by the Performance Profiler to explore the search space of optimization parameters.

Performance Profiler: Performance Profiler carefully collects profile data of the different code versions (each having different loop optimization combinations) for a range of (likely) input sizes, block sizes and packing degrees. Here, the user provides the input sizes in the pragma as a list of expected inputs (N). The profiler varies the block size (starting from input size N) by dividing it by successive powers of 2, while also ensuring that the parameters chosen comply with all the platform-specific resource constraints. For packing degree, starting at 1, it multiplies it with successive powers of 2, also adhering to the resource limitations. This exploration stops when a local minimum is reached. Thus, by essentially performing a sweep across likely input sizes and using relevant combinations of optimization parameters, it collects relevant and sufficient profile data which we then use to construct models for latency and cost.

Latency and Cost Model: As mentioned, the Latency and Cost Models are constructed using the relevant data from the Performance Profiler, thus allowing the framework to rank E2E latency and cost as a function of optimization parameters (as explained in detail in Section 4). In our current approach, we save the top 10 parameter configurations for each input to facilitate future search space exploration. Note that we also perform online updates on the model using current runtime metrics to keep it up-to-date.

Runtime Optimizer: All components discussed thus far mostly work offline (before the actual app execution). When a user provides an input and an optimization objective to our system to run an (already-profiled) app, the Runtime Optimizer picks the best loop optimization and packing parameters to meet the goal by consulting the constructed performance/cost models. For unseen input sizes, the optimizer initiates a search space exploration for this unique input, starting from a 'seed' established by the nearest known

input size, and leveraging our models to arrive at a local minima. To expedite this, the exploration is conducted in parallel (across 100 distinct points).

6 IMPLEMENTATION AND EXPERIMENTAL SETUP

FAASTLOOP is implemented using Python. Our compiler support is implemented using the Python's AST module where we integrated the loop optimizations using previous works like [24, 49, 54] with [51] that converts a VM-based code to an SF version. We evaluated FAASTLOOP on AWS using c5n.2xlarge (8 vCPU with high network bandwidth) EC2 machines and Lambdas with 3400MB.

Workloads: We assessed FAASTLOOP across 4 HPC workloads: i) General Matrix Multiplication(GEMM) [47] (linear algebra), a kernel which is pivotal in many other HPC workloads that multiplies two matrices; ii) N-body Simulations [18] (scientific computing), an algorithm used frequently in astrophysics, simulating gravitational interactions among multiple celestial bodies; iii) Smith-Waterman [41] (bioinformatics), that performs local sequence alignment, crucial for comparing DNA, RNA, or protein sequences and understanding genetic relationships; and iv) Map-Reduce Sort [32], an efficient big data processing kernels used in various application domains.

Evaluated Schemes: We compared FAASTLOOP against the following schemes – NUMPYWREN [47], where a single block size is manually selected for optimal runtime across various HPC applications and used for all other input sizes, ProPack [9], and WISEFUSE [37], where they packed multiple parallel threads in one SF. For each scheme, we chose the most widely adopted parallelization strategy applicable. For NUMPYWREN, we manually calibrated the block size for a smaller input size (N) to optimize performance and cost, while keeping within the constraints of SF resources, and the block size remained fixed for other input sizes. On the other hand, for ProPack/WISEFUSE, we adopted the optimal block size identified from NUMPYWREN and coupled it with adaptive task packing. This configuration was also tested on a larger input size, with adaptive task packing in ProPack/WISEFUSE that adjusts as needed to optimize for the new input size. We combine WISEFUSE and ProPack as one scheme because WISEFUSE also implements task packing in SF as one of its optimizations. As a point of reference, in many experiments, we compare the performance/cost of the concerned scheme with respect to an Oracle scheme to give a notion of how close it is to the ideal choice of parameters. The Oracle scheme is an "optimal parallelization strategy" for a given input size for an app, determined by manually exploring all potential combinations of parameters.

7 EVALUATION

We evaluate FAASTLOOP, and present key results as questions that address critical aspects of its performance and cost efficiency. Unless specified, results shown for performance and cost are separate experiments, where each scheme is tuned for the specific goal.

How well does FAASTLOOP perform compared to other schemes for different applications (scaled across various input sizes), with respect to both latency and cost?

Figure 11 shows both the normalized execution time and cost relative to an Oracle scheme (here 100% refers to the cost/time of Oracle). Below, we analyze the graphs for each workload separately.

GEMM: For the smaller input matrix size (N=2K), we observe that FAASTLOOP achieves similar time and cost efficiency as the Oracle. However, despite manually optimizing blocksize for NUMPYWREN and packing degree for PROPACK/WISEFUSE, for N=2K, both perform 1.5× and 1.75× worse in terms of time and cost, respectively, compared to Oracle. A key reason for FAASTLOOP's efficiency being close to Oracle is its loop optimizations that alter the code structure to suit SF architecture better. The typical code structure employed by most VMs features outer loop parallelism with a k-i-j order. However, when adapting the code for SFs, the structure shifts to middle-loop parallelism with an i-j-k loop order which reduces the task size per lambda and reduces data transfers. Also to note, ProPack/WISEFUSE has a similar outcome to NUMPYWREN. This occurs because the packing optimization falls short; the increase in runtime for each Serverless Function (SF) due to task interference outweighs the advantages of reduced scaling time from lesser number of SFs. As a result, the outcomes are similar to those of NUMPYWREN, which does not utilize this optimization.

Upon scaling the input size to N=8K, FAASTLOOP substantially outperforms its competitors, achieving a 3.3× improvement in end-to-end execution latency when focusing on latency reduction, and a 2.1× reduction in costs when cost minimization is the goal. This notable performance gap arises from the failure of the other schemes to adjust their block sizes or code structures in response to a larger input size. Such lack of adaptation not only affects overall efficiency but also limits the potential benefits that could be derived from the task packing optimization of ProPack/WISEFUSE. This highlights the critical need for customizing parallelization strategies to match varying input sizes in serverless environments.

Nbody Simulation: For smaller input matrices (N=4K), FAASTLOOP and ProPack/WISEFUSE outperform NUMPYWREN, exhibiting almost similar improvements - 1.42× better latency and 1.94× lower cost. The similar performance between FAASTLOOP and ProPack/WISEFUSE indicates that

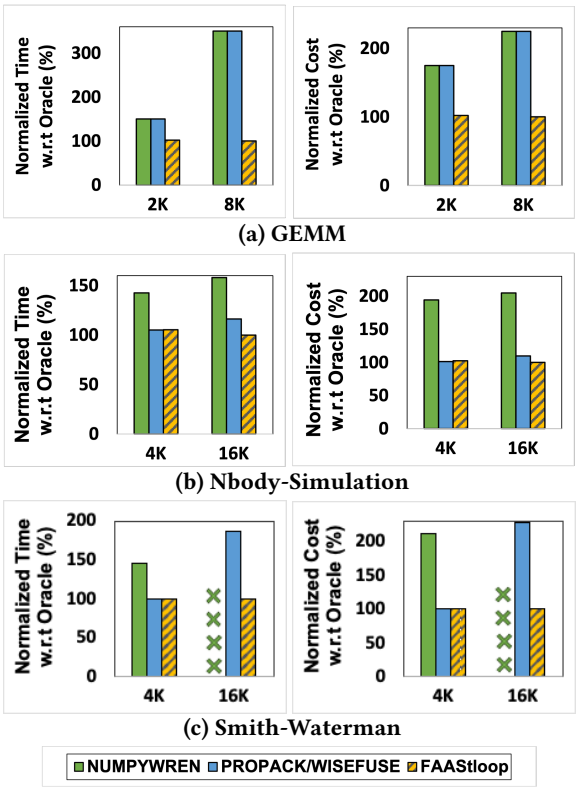


Figure 11: Comparison of normalized time/cost with respect to the Oracle scheme between FAAStloop vs other schemes. The left graph focuses on minimizing latency, whereas the right graph targets cost minimization, across various input sizes (N) on the x-axis. "NUMPYWREN" and "PROPACK/WISEFUSE" were manually optimized for smaller input sizes and tested for larger inputs.

the default code structure for parallelization (which features outer loop parallelism with an i-j loop order and a manually tuned block size of 128) is well-suited for the SF environment as well, with task packing (pd = 4) further enhancing efficiency. But FAASTLOOP picks a different order (same block size as before, but with inner loop parallelism with an i-j loop order, offering finer-grained work division, thus enabling efficient packing (pd=8)) which is slightly faster/cheaper. When scaling up to N=16K, FAASTLOOP continues to outperform NUMPYWREN, showing improvements of 1.58× in time and 2.04× in cost over NUMPYWREN, and also surpasses ProPack/WISEFUSE, albeit by a smaller margin of 16% in time and 9% in cost. The smaller gap with ProPack/-WISEFUSE is attributed to ProPack/WISEFUSE adaptive task packing and already efficient default code structure for this particular scenario.

Smith Waterman: For smaller input (N=4K), adjusting the block size in the default code structure [9, 41] results in hitting the concurrency limit set by AWS Lambda (1000) before reaching the minima. Therefore, we chose a block size that maximizes concurrency, which, consequently, is the lowest achievable time and cost point within our exploration limits. For N=4K, NUMPYWREN is 1.45× slower and 2.12× costlier than FAASTLOOP and ProPack/WISEFUSE. For those schemes, task packing significantly lowers the number of parallel SFs (*numSF*) which, in turn, reduces the cost and overall time (by reducing the scaling time). However, for larger inputs (N=16K), maintaining the same block size and parallelization strategy renders NUMPYWREN infeasible (crosses in Figure 11c), as it surpasses the concurrency limit for this block size and its default code structure. Additionally, despite using task packing optimization, ProPack/WISEFUSE underperforms versus FAASTLOOP, with a 1.86× longer time and 2.27× higher cost because to stay within the concurrency limits, it packs several tasks in each SF, which increases the job interference, thereby increasing the runtime of each SF.

How much does each of the optimizations contribute?

To answer this, we take the GEMM workload as an example and study the resultant improvements of each optimization. As a baseline, we selected the widely used OUTER-LOOP(k) loop structure typical for monolithic (non-serverless) architectures [53, 63]. We picked a block size of 1K and an input size of N=8K. Here, 1K is chosen because it is not the optimal block size for N=8K, thus, allowing the 'blocking/tiling' optimization to show its benefits. It is clear from Figure 12 that loop reordering alone can double the speed of end-to-end execution, indicating that *regardless of adjustment in block size or task packing (as explored in NUMPYWREN and ProPack), the loop order plays a crucial role in reducing time and cost*, especially when scaling to larger input sizes. Another important observation is that the combined effect of all optimizations surpasses the sum of the individual improvements (yielding a 3× speedup). This suggests that *certain optimization opportunities only emerge after other optimizations are applied.* For instance, task packing may not significantly speed up the process on its own, but once combined with other optimizations, it contributes greatly to reducing the overall service time.

We can see that relative cost, which primarily depends on the number of parallel SFs (*numSF*) and SF runtime, is reduced by blocking (more significantly) and task packing. Both blocking and packing achieve this by lowering *numSF* while balancing the runtime/task interference effectively. Again, the final optimization, which combines loop optimizations and task packing, results in greater cost savings (2.2×) than the individual optimizations alone.

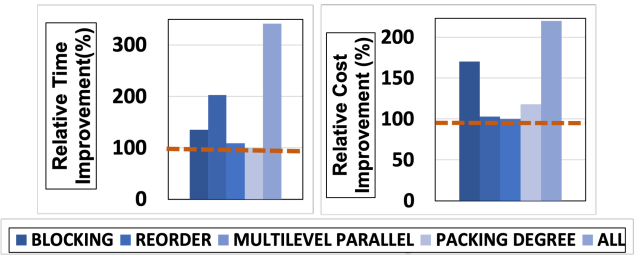


Figure 12: Relative improvements brought by individual optimization over the baseline for minimizing latency(left) and cost(right) for GEMM(N=8K). Baseline:OUTERLOOP(k) with BLK=1K and PD=1.

How does FAASTLOOP perform, given a time or cost tolerance?

FAASTLOOP can adapt its optimization parameters where user goals are defined to minimize cost within a specified time tolerance, or conversely, to minimize time within a set cost tolerance. In this case, we set tolerance margins of 10%, 20%, and 50% above FAASTLOOP's minimum time/cost for the respective cost/time budget. This approach aims to assess how much the secondary metric (cost or time) can be optimized within these defined tolerance limits. Note that the time and cost values shown *within* each plot here pertain to the same experiment.

For the GEMM workload with an input size of N=4K, depicted in Figure 13, relaxing FAASTLOOP's minimum time tolerance margin results in substantial cost savings. For instance, providing a 20% margin over the minimum achievable end-to-end time results in a (substantial) 94% drop in cost (bringing it from being 2.25× to just 29% more than the Oracle), with only a (minimal) 12% increase in time. For a 50% margin over the minimum time, shows a larger time increase (about 40%), but the cost doesn't decrease much more than it did at the 20% time budget, standing at 105%. Similarly, setting a cost budget by providing a margin over the minimum time yields a considerable reduction in end-to-end time. For instance, with a 50% margin, there is a substantial (113%) reduction in time compared to the minimum cost, albeit at a 29% increase in cost. Thus, using FAASTLOOP, users can opt for a minimal increase in one metric (time or cost) over the minimum, to yield significant improvements in the other.

What is the practicality of offloading applications to SFs via FAASTLOOP versus using large VMs?

We now examine the feasibility of offloading n-body simulation with a 16K input size using FAASTLOOP compared to running it on a large VM (specifically a 96vcpu VM, which takes the same time to execute the app as FAASTLOOP for a single request). This analysis is illustrated in Figure 15. It can be observed that offloading to a VM becomes more sensible only if the hourly request rate exceeds 53. Below this

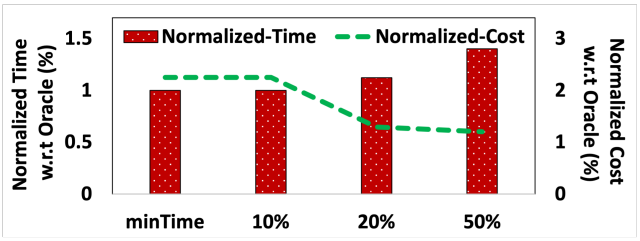


Figure 13: Impact of increasing latency tolerance on cost for GEMM with N=4K. These results demonstrate how relaxing FAASTLOOP's minimum time margin leads to considerable cost reductions.

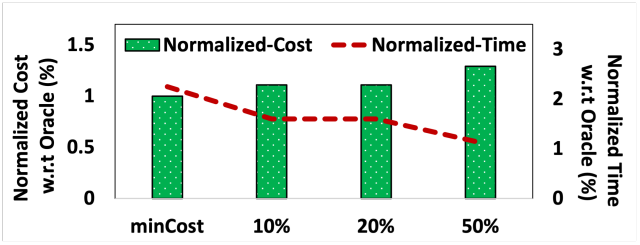


Figure 14: Impact of increasing cost tolerance on latency for GEMM with N=4K. These results demonstrate how relaxing FAASTLOOP's minimum cost margin leads to considerable time improvement.

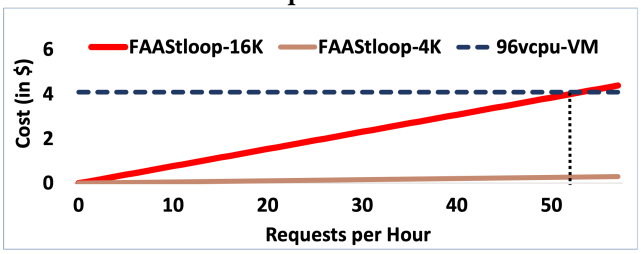


Figure 15: Breakeven analysis of FAASTLOOP vs. a Large 96vcpu VM for N-body Simulation of N=16K. Also shown N=4K."

threshold, it is more advantageous to run the apps on SFs using FAASTLOOP. As noted in the paper [46], in cloud centers, 81% of SF invocations typically occur less than 60 times per hour (which aligns with the above observation). We also observe that running the n-body simulation with a smaller input size (4K) on FAASTLOOP is significantly cheaper, allowing for a much higher threshold till which FAASTLOOP is preferred over the VM execution. Therefore, if there is a variation in the input size, FAAStloop can handle higher request rates as well. A hybrid VM-FAAStloop can also be a strategic choice, where a smaller VM is constantly operational to handle frequent requests of smaller input sizes, while larger, less frequent input sizes are offloaded to FAAStloop.

How does FAAStloop compare with Oracle?

When comparing FAASTLOOP to Oracle, we conducted the experiments using the n-body simulation and GEMM

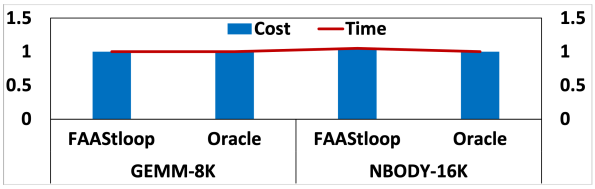


Figure 16: Comparison of FAASTLOOP and Oracle for N-Body Simulation(N=16K) and GEMM(N=8K).FAASTLOOP's performance closely matches with a maximum deviation of ~5% in both time and cost for n-body simulation

workloads. Note that Oracle uses manually tuned (optimal) combinations of optimization parameters found through an exhaustive exploration of the parameter search space. The results here are particularly striking, with very little difference in cost/time observed between FAASTLOOP and Oracle (Figure 16). For instance, the most deviation between the two schemes occurs for n-body simulation, where the difference (in terms of both execution time and cost) between FAASTLOOP and the Oracle is ~5%. This is a testament to the accuracy of the models used by FAASTLOOP and its search space exploration strategy.

Is FAASTLOOP also effective and scalable for single-loop apps? Many HPC apps frequently utilize the map-reduce paradigm, which is often structured around a single-loop design [9]. Two out of four optimizations (loop reorder and multi-level parallelism) that FAASTLOOP applies depend on the presence of a nested loop. Thus, for single loop apps, only loop blocking and task packing would be FAASTLOOP's viable optimizations. To understand how well FAASTLOOP performs with only two optimizations against other schemes, we evaluate it on a Map Reduce Sort (henceforth, simply called Sort) [9, 60] workload by using input sizes starting at 0.5M and scaled to 33M. Consider Figure 17. From the end-to-end time graph (normalized with respect to FAASTLOOP), we observe that NUMPYWREN performs significantly worse (over 3.1× for N = 8M and over 6.5× for N = 33M) than FAASTLOOP. This is because NUMPYWREN has a fixed block size for all inputs (unless manually calibrated again for a new input) and does not employ adaptive task packing. This observation is consistent with our results for nested loop apps. ProPack/WISEFUSE scales better than NUMPYWREN due to its adaptive task packing optimization, but even then, we see as the input size increases, it performs significantly worse (over 1.13× for N = 8M and over 1.43× for N = 33M) than FAASTLOOP.

The normalized cost graph also follows a similar trend, albeit scaling more gradually. One interesting point to note is that the cost difference between ProPack/WISEFUSE and FAASTLOOP initially decreases (as the input size goes from 0.5M to 8M) and later increases (up to 1.12×). This fluctuation

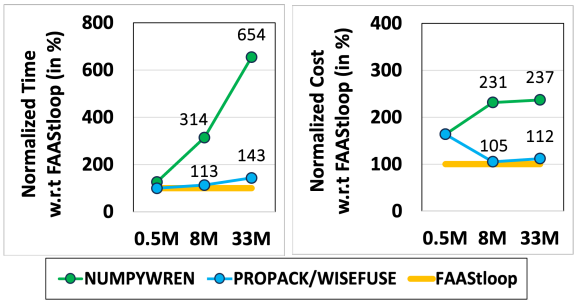


Figure 17: Sort (Single-Loop) application: Time/cost of all schemes normalized with respect to that of FAASTLOOP.

is linked to the initial block size setting for NUMPYWREN, which is manually tuned to 0.5M (the most cost-effective setting without task packing). Since ProPack/WISEFUSE's block size is dependent on the one used for NUMPYWREN, it starts with a block size of 0.5M too, leading to only one parallel serverless function (SF) being used and no chance for task packing to reduce costs. Consequently, at 0.5M, ProPack/WISEFUSE's inability to utilize task packing makes it less cost-efficient. However, as the input size increases beyond 0.5M, the number of SFs grows, allowing ProPack/-WISEFUSE to employ task packing and optimize costs more effectively. Nevertheless, FAASTLOOP demonstrates superior scalability in comparison to *both* NUMPYWREN and ProPack/WISEFUSE, even with a limited set of optimizations available to it. As observed, with the growth in input size, the performance gap between FAAStloop and the others widens, highlighting FAAStloop's robust scaling capabilities.

8 DISCUSSION

In the previous sections, we explored how FAASTLOOP enhances loop-based applications using an offline-online profiler to train latency and cost models. This method requires roughly 100 data points from real-world system evaluations, which incurs certain costs during the profiling stage. However, it's important to note that this is a one-time expense, recoverable within just a few tens of requests, which should typically be a much smaller fraction compared to the number of times the app will be used in a total lifetime. Since different applications could have different parameters and runtimes, the profiling cost for each application is different. For instance, DNA sequencing requires about 75 requests to offset the $10 cost of offline profiling and N-body Simulation needs about 36 requests to cover a $4 profiling expense.

Our modeling approach initially relies on offline profiling and is followed by regular online updates to account for fluctuations in usage patterns. While the approach is comprehensive, it could be complex and add additional overhead. This overhead may become problematic if the total number of requests is too low to offset the costs of offline profiling.

Since we lacked prior data on users' previous requests, we resorted to offline-online profiling. However, in real-world applications, we anticipate profiling to occur in real-time with actual user-requests. This could help us to transition to a fully online profiling approach, which could substantially reduce complexity, and significantly save the overheads of time and costs, as these requests would occur regardless. We expect these costs to be minimal (in most cases less than 5% additional cost), making the approach more scalable.

Additionally, in terms of generalizability of this work, we would also like to highlight that we have explored up to 3 nested loop applications, but if the apps have longer nested loops, it could increase the parameter search space requiring >100 runs for the model to converge, thereby potentially increasing the offline profiling cost.

9 CONCLUDING REMARKS

This paper presents FAASTLOOP, a user-side framework that deploys HPC applications on a serverless platform using, for the first time, loop optimizations and task packing to (almost always) optimally divide tasks amongst serverless functions. Due to being able to predict the potential impact of the choice of optimization parameters on the cost/latency using its analytical models, FAASTLOOP can dynamically adapt its parameters for different inputs to applications, unlike prior works. By virtue of these features, FAASTLOOP outperforms state-of-the-art frameworks by up to 3.3× and 2.1×, in terms of end-to-end execution latency and cost, respectively.

10 ACKNOWLEDGEMENT

We are indebted to our anonymous reviewers and shepherd, Vaibhav Arora, for their insightful comments. This research was partially supported by NSF grants #2116962, and #1931531. All product names used here are for identification purposes only and may be trademarks of their respective companies.

REFERENCES

[1] 2021. Cloud Adoption is Driving HPC Toward Digital R&D. https://bigcompute.org/blog/cloud-adoption-is-driving-hpc-toward-digital-rd/.

[2] 2022. AWS Lambda Pricing. https://aws.amazon.com/lambda/pricing/.

[3] 2023. State of Serverless 2023 Report Suggests Increasing Serverless Adoption. https://www.infoq.com/news/2023/09/state-serverless-report/.

[4] 2023. Survey Reports Rise of the Cloud for Engineering Simulation Workloads. https://www.ansys.com/blog/survey-reports-rise-of-cloud-adoption/.

[5] 2023. The State of Serverless. https://www.datadoghq.com/state-of-serverless/.

[6] Lixiang Ao, Liz Izhikevich, Geoffrey M. Voelker, and George Porter. 2018. Sprocket: A Serverless Video Processing Framework. In *Proceedings of the ACM Symposium on Cloud Computing* (Carlsbad, CA, USA) *(SoCC '18)*. Association for Computing Machinery, New York, NY, USA, 263–274. https://doi.org/10.1145/3267809.3267815

[7] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. 1994. Compiler transformations for high-performance computing. *ACM Comput. Surv.* 26, 4 (dec 1994), 345–420. https://doi.org/10.1145/197405.197406

[8] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. *Research advances in cloud computing* (2017), 1–20.

[9] Rohan Basu Roy, Tirthak Patel, Richmond Liew, Yadu Nand Babuji, Ryan Chard, and Devesh Tiwari. 2023. ProPack: Executing Concurrent Serverless Functions Faster and Cheaper. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing*. 211–224.

[10] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying Parallel and Distributed Deep Learning: An In-depth Concurrency Analysis. *ACM Comput. Surv.* 52, 4, Article 65 (Aug. 2019), 43 pages. https://doi.org/10.1145/3320060

[11] Vivek M. Bhasi, Jashwant Raj Gunasekaran, Aakash Sharma, Mahmut Taylan Kandemir, and Chita Das. 2022. Cypress: Input Size-Sensitive Container Provisioning and Request Scheduling for Serverless Platforms. In *Proceedings of the 13th Symposium on Cloud Computing* (San Francisco, California) *(SoCC '22)*. Association for Computing Machinery, New York, NY, USA, 257–272. https://doi.org/10.1145/3542929.3563464

[12] Vivek M. Bhasi, Aakash Sharma, Shruti Mohanty, Mahmut Taylan Kandemir, and Chita R. Das. 2024. Paldia: Enabling SLO-Compliant and Cost-Effective Serverless Computing on Heterogeneous Hardware. In *2024 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 100–113. https://doi.org/10.1109/IPDPS57955.2024.00018

[13] Sandeepa Bhuyan, Ziyu Ying, Mahmut T. Kandemir, Mahanth Gowda, and Chita R. Das. 2024. GameStreamSR: Enabling Neural-Augmented Game Streaming on Commodity Mobile Platforms. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 1309–1322. https://doi.org/10.1109/ISCA59077.2024.00097

[14] Sandeepa Bhuyan, Shulin Zhao, Ziyu Ying, Mahmut T. Kandemir, and Chita R. Das. 2022. End-to-end Characterization of Game Streaming Applications on Mobile Platforms. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 1, Article 10 (Feb. 2022), 25 pages. https://doi.org/10.1145/3508030

[15] Zhi Chen, Zhangxiaowen Gong, Justin Josef Szaday, David C. Wong, David Padua, Alexandru Nicolau, Alexander V. Veidenbaum, Neftali Watkinson, Zehra Sura, Saeed Maleki, Josep Torrellas, and Gerald DeJong. 2017. LORE: A loop repository for the evaluation of compilers. In *2017 IEEE International Symposium on Workload Characterization*. Institute of Electrical and Electronics Engineers Inc., 219–228. https://www.microsoft.com/en-us/research/publication/lore-a-loop-repository-for-the-evaluation-of-compilers/

[16] Tamara Dancheva, Unai Alonso, and Michael Barton. 2023. Cloud benchmarking and performance analysis of an HPC application in Amazon EC2. *Cluster Computing* (2023), 1–18.

[17] Simon Eismann, Long Bui, Johannes Grohmann, Cristina Abad, Nikolas Herbst, and Samuel Kounev. 2021. Sizeless: predicting the optimal size of serverless functions. In *Proceedings of the 22nd International Middleware Conference* (Québec city, Canada) *(Middleware '21)*. Association for Computing Machinery, New York, NY, USA, 248–259. https://doi.org/10.1145/3464298.3493398

[18] Erich Elsen, Vaidyanathan Vishal, Mike Houston, Vijay S. Pande, Pat Hanrahan, and Eric Darve. 2007. N-Body Simulations on GPUs. *CoRR* abs/0706.3060 (2007). arXiv:0706.3060 http://arxiv.org/abs/0706.3060

[19] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From

Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[20] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. 2019. From Laptop to Lambda: Outsourcing Everyday Jobs to Thousands of Transient Functional Containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*. USENIX Association, Renton, WA, 475–488. http://www.usenix.org/conference/atc19/presentation/fouladi

[21] Evangelos Georganas, Jorge González-Domínguez, Edgar Solomonik, Yili Zheng, Juan Tourino, and Katherine Yelick. 2012. Communication avoiding and overlapping for numerical linear algebra. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 1–11.

[22] Google Cloud. 2021. What are the limits of serverless for online gaming. https://serialized.net/2021/03/serverless_gaming_limits/

[23] Google Cloud. 2024. Google Cloud Functions Pricing. https://cloud.google.com/functions/pricing

[24] Serge Guelton, Pierrick Brunet, Mehdi Amini, Adrien Merlini, Xavier Corbillon, and Alan Raynaud. 2015. Pythran: enabling static optimization of scientific Python programs. *Computational Science Discovery* 8, 1 (mar 2015), 014001. https://doi.org/10.1088/1749-4680/8/1/014001

[25] Giulia Guidi, Marquita Ellis, Aydin Buluç, Katherine Yelick, and David Culler. 2021. 10 Years Later: Cloud Computing is Closing the Performance Gap. In *Companion of the ACM/SPEC International Conference on Performance Engineering* (Virtual Event, France) *(ICPE '21)*. Association for Computing Machinery, New York, NY, USA, 41–48. https://doi.org/10.1145/3447545.3451183

[26] Emna Hammami and Yosr Slama. 2017. An Overview on Loop Tiling Techniques for Code Generation. In *2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (AICCSA)*. 280–287. https://doi.org/10.1109/AICCSA.2017.168

[27] Gazi Karam Illahi, Thomas Van Gemert, Matti Siekkinen, Enrico Masala, Antti Oulasvirta, and Antti Ylä-Jääski. 2020. Cloud Gaming with Foveated Video Encoding. *ACM Trans. Multimedia Comput. Commun. Appl.* 16, 1, Article 7 (Feb. 2020), 24 pages. https://doi.org/10.1145/3369110

[28] Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. 2018. Serving deep learning models in a serverless platform. In *2018 IEEE International conference on cloud engineering (IC2E)*. IEEE, 257–262.

[29] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, and Chita R. Das. 2023. Optimizing CPU Performance for Recommendation Systems At-Scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 77, 15 pages. https://doi.org/10.1145/3579371.3589112

[30] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[31] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, et al. 2019. Cloud programming simplified: A berkeley view on serverless computing. *arXiv preprint arXiv:1902.03383* (2019).

[32] Ana Klimovic, Yawen Wang, Christos Kozyrakis, Patrick Stuedi, Jonas Pfefferle, and Animesh Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 789–794. https://www.usenix.org/conference/atc18/presentation/klimovic-serverless

[33] Jakub Kurzak, Stanimire Tomov, and Jack Dongarra. 2012. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems* 23, 11 (2012), 2045–2057. https://doi.org/10.1109/TPDS.2011.311

[34] Hyungro Lee, Kumar Satyam, and Geoffrey Fox. 2018. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 442–450.

[35] Yongkang Li, Yanying Lin, Yang Wang, Kejiang Ye, and Chengzhong Xu. 2023. Serverless Computing: State-of-the-Art, Challenges and Opportunities. *IEEE Transactions on Services Computing* 16, 2 (2023), 1522–1539. https://doi.org/10.1109/TSC.2022.3166553

[36] Wes Lloyd, Shruti Ramesh, Swetha Chinthalapati, Lan Ly, and Shrideep Pallickara. 2018. Serverless computing: An investigation of factors influencing microservice performance. In *2018 IEEE international conference on cloud engineering (IC2E)*. IEEE, 159–169.

[37] Ashraf Mahgoub, Edgardo Barsallo Yi, Karthick Shankar, Eshaan Minocha, Sameh Elnikety, Saurabh Bagchi, and Somali Chaterji. 2022. Wisefuse: Workload characterization and dag transformation for serverless workflows. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 6, 2 (2022), 1–28.

[38] Garrett McGrath and Paul R Brenner. 2017. Serverless computing: Design, implementation, and performance. In *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*. IEEE, 405–410.

[39] Microsoft Azure. 2024. Azure Functions Pricing. https://azure.microsoft.com/en-us/pricing/details/functions/

[40] Girish Mururu, Sharjeel Khan, Bodhisatwa Chatterjee, Chao Chen, Chris Porter, Ada Gavrilovska, and Santosh Pande. 2023. Beacons: An End-to-End Compiler Framework for Predicting and Utilizing Dynamic Loop Characteristics. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 228 (oct 2023), 31 pages. https://doi.org/10.1145/3622803

[41] Xingzhi Niu, Dimitar Kumanov, Ling-Hong Hung, Wes Lloyd, and Ka Yee Yeung. 2019. Leveraging Serverless Computing to Improve Performance for Sequence Comparison. In *Proceedings of the 10th ACM International Conference on Bioinformatics, Computational Biology and Health Informatics* (Niagara Falls, NY, USA) *(BCB '19)*. Association for Computing Machinery, New York, NY, USA, 683–687. https://doi.org/10.1145/3307339.3343465

[42] Rohan Basu Roy, Tirthak Patel, Vijay Gadepally, and Devesh Tiwari. 2022. Mashup: making serverless computing useful for HPC workflows via hybrid execution. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 46–60. https://doi.org/10.1145/3503221.3508407

[43] Iman Sadooghi, Jesús Hernández Martin, Tonglin Li, Kevin Brandstatter, Ketan Maheshwari, Tiago Pais Pitta de Lacerda Ruivo, Gabriele Garzoglio, Steven Timm, Yong Zhao, and Ioan Raicu. 2017. Understanding the Performance and Potential of Cloud Computing for Scientific Applications. *IEEE Transactions on Cloud Computing* 5, 2 (2017), 358–371. https://doi.org/10.1109/TCC.2015.2404821

[44] Josep Sampé, Gil Vernik, Marc Sánchez-Artigas, and Pedro García-López. 2018. Serverless Data Analytics in the IBM Cloud. In *Proceedings of the 19th International Middleware Conference Industry* (Rennes, France) *(Middleware '18)*. Association for Computing Machinery, New York, NY, USA, 1–8. https://doi.org/10.1145/3284028.3284029

[45] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, Joao Carreira, Neeraja J. Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. 2021. What Serverless Computing is and Should Become: The next Phase of Cloud Computing. *Commun.*

ACM 64, 5 (apr 2021), 76–84. https://doi.org/10.1145/3406011

[46] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[47] Vaishaal Shankar, Karl Krauth, Kailas Vodrahalli, Qifan Pu, Benjamin Recht, Ion Stoica, Jonathan Ragan-Kelley, Eric Jonas, and Shivaram Venkataraman. 2020. Serverless linear algebra. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 281–295.

[48] Aakash Sharma, Vivek M. Bhasi, Sonali Singh, Rishabh Jain, Jashwant Raj Gunasekaran, Subrata Mitra, Mahmut Taylan Kandemir, George Kesidis, and Chita R. Das. 2023. Stash: A Comprehensive Stall-Centric Characterization of Public Cloud VMs for Distributed Deep Learning. In *2023 IEEE 43rd International Conference on Distributed Computing Systems (ICDCS)*. 1–12. https://doi.org/10.1109/ICDCS57875.2023.00023

[49] David Sheffield, Michael Anderson, and Kurt Keutzer. 2012. Automatic generation of application-specific accelerators for FPGAs from python loop nests. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*. 567–570. https://doi.org/10.1109/FPL.2012.6339372

[50] Edgar Solomonik and James Demmel. 2011. Communication-optimal parallel 2.5 D matrix multiplication and LU factorization algorithms. In *European Conference on Parallel Processing*. Springer, 90–109.

[51] Myungjun Son, Shruti Mohanty, Jashwant Raj Gunasekaran, Aman Jain, Mahmut Taylan Kandemir, George Kesidis, and Bhuvan Urgaonkar. 2022. Splice: An Automated Framework for Cost-and Performance-Aware Blending of Cloud Services. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*. IEEE, 119–128.

[52] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, Mauro Bianco, Bradford L Chamberlain, Romain Cledat, H Carter Edwards, Hal Finkel, et al. 2017. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 10 (2017), 3007–3020.

[53] Richard Michael Veras, Tze Meng Low, Tyler Michael Smith, Robert van de Geijn, and Franz Franchetti. 2016. Automating the last-mile for high performance dense linear algebra. *arXiv preprint arXiv:1611.08035* (2016).

[54] Neftali Watkinson, Aniket Shivam, Alexandru Nicolau, and Alexander Veidenbaum. 2019. Teaching Parallel Computing and Dependence Analysis with Python. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 320–325. https://doi.org/10.1109/IPDPSW.2019.00061

[55] Tobias Weinzierl. 2021. *OpenMP Primer: BSP on Multicores*. Springer International Publishing, Cham, 187–210. https://doi.org/10.1007/978-3-030-76194-3_16

[56] Jinfeng Wen, Zhenpeng Chen, Xin Jin, and Xuanzhe Liu. 2023. Rise of the planet of serverless computing: A systematic review. *ACM Transactions on Software Engineering and Methodology* 32, 5 (2023), 1–61.

[57] R.C. Whaley and J.J. Dongarra. 1998. Automatically Tuned Linear Algebra Software. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*. 38–38. https://doi.org/10.1109/SC.1998.10004

[58] Michael E Wolf and Monica S Lam. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel & Distributed Systems* 2, 04 (1991), 452–471.

[59] Jingling Xue. 2000. *Loop tiling for parallelism*. Vol. 575. Springer Science & Business Media.

[60] Owen O'Malley Yahoo! 2008. Terabyte Sort on Apache Hadoop. (Mai 2008 2008). http://www.hpl.hp.com/hosted/sortbenchmark/YahooHadoop.pdf

[61] Qing Yi and Ken Kennedy. 2004. Improving memory hierarchy performance through combined loop interchange and multi-level fusion. *The International Journal of High Performance Computing Applications* 18, 2 (2004), 237–253.

[62] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. 2016. YASK—Yet Another Stencil Kernel: A framework for HPC stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*. IEEE, 30–39.

[63] Huaqing Zhang, Xiaolin Cheng, Hui Zang, and Dae Hoon Park. 2019. Compiler-level matrix multiplication optimization for deep learning. *arXiv preprint arXiv:1909.10616* (2019).

[64] Hong Zhang, Yupeng Tang, Anurag Khandelwal, Jingrong Chen, and Ion Stoica. 2021. Caerus: NIMBLE Task Scheduling for Serverless Analytics. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 653–669. https://www.usenix.org/conference/nsdi21/presentation/zhang-hong

[65] Miao Zhang, Yifei Zhu, Cong Zhang, and Jiangchuan Liu. 2019. Video processing with serverless computing: a measurement study. In *Proceedings of the 29th ACM Workshop on Network and Operating Systems Support for Digital Audio and Video* (Amherst, Massachusetts) *(NOSSDAV '19)*. Association for Computing Machinery, New York, NY, USA, 61–66. https://doi.org/10.1145/3304112.3325608

[66] Wen Zhang, Vivian Fang, Aurojit Panda, and Scott Shenker. 2020. Kappa: a programming framework for serverless computing. In *Proceedings of the 11th ACM Symposium on Cloud Computing* (Virtual Event, USA) *(SoCC '20)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/3419111.3421277