Tackling Virtual Infrastructure Allocation in Cloud Data Centers: a GPU-Accelerated Framework

Lucas L. Nesi[◇], Maurício A. Pillon[◇], Marcos D. de Assunção[⊕], Charles C. Miers[◇], Guilherme P. Koslovski[◇]
Graduate Program in Applied Computing – Santa Catarina State University – Joinville – Brazil[◇]
Inria Avalon, LIP Laboratory, ENS Lyon, University of Lyon – France[⊕]
lucas.nesi@edu.udesc.br, mauricio.pillon@udesc.br,
assuncao@acm.org, charles.miers@udesc.br, guilherme.koslovski@udesc.br

***Abstract*—Allocating IT resources to Virtual Infrastructures (VIs) (*i.e.,* groups of VMs, virtual switches, and their network interconnections) is a problem which belongs to a class known to be NP-hard. Several approaches designed to run on CPUs have been proposed for reducing the search space and finding suitable allocation solutions. Most algorithms, however, face scalability issues when considering current cloud data centers comprising thousands of servers. To overcome these limitations, this work offers a set of allocation algorithms designed for Graphic Processing Units (GPUs). Experimental results evaluate the scalability of the algorithms and demonstrate their ability to handle three large-scale data center topologies.**

1. Introduction

Under the Cloud computing model, Infrastructure-as-a-Service (IaaS) providers deliver on-demand access to VIs comprising virtual resources that are deployed onto Data Center (DC) infrastructure. Allocating a VI consists in meeting Quality-of-Service (QoS) requirements of Virtual Machines (VMs) and provisioning virtual network (*i.e.,* switches and links) to tenants. Each tenant can customize a VI according to the requirements of an application whose performance can be influenced by the network configuration and assigned resources.

Cloud providers rely on complex mechanisms to allocate physical resources for hosting VIs. The allocation task can be viewed as a graph embedding problem [1]. Given a graph $G^v = (N^v, L^v)$ representing a VI request (where N^v are virtual nodes, switches and VMs, and L^v are virtual links) and a graph $G^s = (N^s, L^s)$ representing the cloud data center (N^s is composed of servers and data forwarding resources, and L^s denotes physical links), the problem is to find a mapping function $f(G^v)$ injector in G^s, respecting the QoS for vertices and edges.

An allocation solution can optimize a single or multiple criteria such as decreasing DC load and VI latency, reducing energy consumption, among others [1]. Such problem falls into a class known to be NP-hard [2]. The number of possible allocation solutions for a given VI request grows with the size of the VI and DC graphs, which is an issue as modern DC architectures, such as Fat-Tree [3], BCube [4], and DCell [5] can have thousands of servers and links. Existing solutions are little applicable to real cloud DCs due to their CPU-driven design usually combined with topology-oriented pruning techniques.

GPUs offer a high-degree of parallelism, high throughput memory and general purpose programming tools, such as Compute Unified Device Architecture (CUDA) [6]. Designing allocation algorithms for GPUs, however, is not trivial as well-known graph algorithms must be refactored to run efficiently. Moreover, the literature on VI allocation is vast and contains algorithms for isomorphism identification, vertex and edge ranking [7], data clustering [8], and path selection [9]. Refactoring is essential as most traditional graph algorithms employ techniques designed to run on CPUs, such as backtracking, which are inefficient on GPU architecture. As GPU Single Instruction Multiple Data (SIMD) operations execute in groups of threads, the occurrence of a single operation that diverges from others can result in lost parallelism.

Therefore, this work makes two main contributions: *(i)* We refactor a set of graph algorithms that are then used to compose allocation strategies as part of a GPU-accelerated framework for VI allocation. Microbenchmarks are used to compare the speedup of each individual algorithm against CPU-based approaches. The results show the applicability of GPU algorithms to large-scale cloud data centers. *(ii)* Performance evaluation results demonstrate the benefits of massively parallel processing for VI allocation using the strategies proposed in the framework. The framework, examples, and documentation are publicly available. [1]

The rest of this paper is organized as follows. §2 reviews the literature and challenges on VI allocation. §3 defines the GPU-tailored algorithms for the VI allocation problem. Proof-of-concept allocation mechanisms build from the GPU-based framework are presented in §4 and simulation results are discussed in §5. §6 concludes the paper.

2. Background and Related Work

2.1. Network Topologies of Cloud Data Centers

Data center topologies are commonly designed to support thousands of servers offering high bandwidth and low latency. By virtualizing the physical hardware of servers and network devices DCs can host a large number of applications

1. Available at https://bitbucket.org/lucasnesi/vnegpu.

978-3-903176-14-0

and VIs [10], [11]. Current DC topologies are designed for supporting large-scale networking virtualization techniques [11] where each topology has specific characteristics that can influence VI placement.

DCell is a recursively built topology that provides high bandwidth and scales doubly exponentially as the node degree increases. It reduces the deployment cost by using mini-switches to scale up instead of high-end switches [5], and has a general structure that interconnects basic units [12]. A DCell construction starts with a DCell_0: a single mini-switch that interconnects n servers. At a DCell_1, $n+1$ DCell_0s are linked following a connection function ρ_L. In general, a DCell_k is recursively composed of $t_{k-1}+1$ DCell_{k-1}s, where t_{k-1} denotes the number of vertices on DCell_{k-1}. Each server has a specific identification at each DCell level. At $k>0$ level, each server is identified by $[a,b]$, where a is the DCell_{k-1} and b is the internal identifier.

BCube relies on shipping-containers, a modular data center topology structured as a hyper-cube [4]. Similar to DCell, it uses mini-switches for interconnecting servers, but servers also act on data forwarding. The topology is recursively composed. Initially, a BCube_0 comprises n servers connected to an n-port mini-switch. A BCube_1 is the union of n BCube_0s, connected to n mini-switches. In general, a BCube_k is composed of n $\text{BCube}_{(k-1)}$s and n^k n-ports switches. Concerning the VI allocation onto a BCube, two properties are worth mentioning: *(i)* the shortest path between servers is $2(k+1)$ in the worst case, and *(ii)* different throughput can be observed between communicating servers when an intermediate node acts on data forwarding.

Fat-Tree is a specialized Clos network [3] that uses multiple low-cost commodity k-port switches for interconnecting servers. A centralized network controller handles dynamic data forwarding and manages all switches [10]. The design results in full bisection bandwidth between pods (the basic building block), where each pod contains two layers of $k/2$ switches (aggregation and edge). The aggregation switches are interconnected to core switches (there are $(k/2)^2$ core switches). A Fat-Tree supports up to $k^3/4$ servers, and some properties related to virtual resource provisioning are: *(i)* independently of k, the distance between servers from different pods is 6 hops, and *(ii)* the number of paths between servers from different pods is the number of core switches, $(k/2)^2$.

2.2. CPU-based Algorithms for VI Allocation

Mapping VMs, switches, and links to physical DC resources while respecting QoS constraints and meeting a provider's allocation goal is an NP-hard problem [2]. The state-of-the-art provides Virtual Network Embedding (VNE) and cloud solutions for this mapping problem.

Mixed Integer Program (MIP) or Linear Program (LP) techniques offer optimal solutions that are generally used as baseline for comparisons [13], [9], but the problem complexity and search space often create opportunities for heuristic-based solutions [1]. [13] created two heuristics for joint allocation of virtual processing and communication resources designed to run on CPUs and the simulation scenarios considered a limited set of candidate resources.

Previous work also proposed grouping techniques for reducing the search space or problem complexity [8], [14]. Their scalability and application to real DCs, however, remain a challenge as aggressive pruning of physical and virtual candidates may lead to partial and inefficient solutions under certain topologies [14]. [15] investigated a topology-aware approach to reconfigure and allocate initially rejected requests, whereas [8] focused on a subset of the VNE problem, the virtual cluster allocation, proposing a polynomial-time algorithm based on hose model.

Cloud simulators are also used mainly to assist in algorithm analysis. The CloudSim simulator, for instance, has been widely used for analyzing the scheduling of VMs [16] and virtual networking on cloud DCs. Simulators, however, suffer from slow execution time and scalability issues, thus preventing the evaluation of very large scenarios.

We argue that GPU-accelerated algorithms can be successfully applied to speed up heuristics and simulators, hence allowing for evaluating large-scale scenarios that more closely match the requirements of real cloud DCs. Therefore, §3 summarizes a set of GPU algorithms for VI allocation. The microbenchmark analysis (§3.6) combined with the discussion on allocation policies (§5) highlight how the GPU-accelerated algorithms advance the state-of-the-art.

3. GPU-Based Algorithms for VI Allocation

3.1. Data Structure and Representation on GPUs

The large amount of data needed to represent cloud DC and VI requests can impact the allocation speedup. As a DC graph and a VI request have common characteristics, and both are sparse graphs, we extend the Compressed Sparse Row (CSR) format [17] for representing both graphs. CSR stores the graph topology using a directed edge representation enabling random access on edges of any node. As a VI is commonly formulated as an undirected graph [1], we enhance CSR by using the Edge Map (EM) vector for identifying the source and destination of undirected graphs. The access of edges on directed or undirected graphs is hence performed with $O(1)$ time complexity. Under the proposed scheme, tree vectors are used, namely Source Offset (SO), Destination Indices (DI), and EM. SO's length is given by the number of vertices plus one, and DI's length is twice the number of undirected edges. The graph structure is driven by the edges ij. For a given node i, SO$[i]$ contains the index in DI that has the first target node j whereas the index SO$[i+1]-1$ in DI contains the last target. Hence DI[SO$[i]$] and DI[SO$[i+1]-1$] store, respectively, the first and the final target nodes from ij pairs. For accessing an edge variable for a directed edge k, it is necessary to get the undirected id of the edge, given by EM$[k]$.

3.2. Algorithms for Comparing Edges and Vertices

Page Rank (PR) is commonly used to rank the nodes of a graph based on a given metric and its edges. It is recurrently adapted and used on GPU [18]. For applying

the GPU-accelerated PR on VI allocation, we integrate an on-the-fly definition of a user-specified function to calculate the node rank, as well as an adaptation of the power iteration method to estimate the eigenvalues of a matrix. **Local Resource Capacity (LRC)** [19] can be adapted to a GPU by executing their core function in parallel for different edges and vertices. Kernels can be prepared and scheduled such that each group of threads can achieve coalesced memory access. **Best-Fit (BF)** & **Worst-Fit (WF)**, the GPU implementation iterates over VI nodes, matching each VI node to the most appropriated DC either in a BF or WF fashion. This step runs on the GPU using a single kernel called at each VI node, performing a reduction over the DC nodes.

3.3. Shortest Path Algorithms

Dijkstra. This iterative algorithm checks at each round the frontier nodes (whose minimum distance was updated and can change the distance of other nodes). Our version for GPUs resembles approaches found in the literature [20], however we also need to know all the shortest paths (in number of hops), and not only the minimum distance. The GPU method consists of visiting all the frontier nodes at different parallel threads. Hence the maximum number of iterations is the longest minimum path between all nodes. On modern DC topologies (§2.1), the longest path is fixed with a small value (*e.g.*, 6 for a Fat-Tree). Formally, the GPU implementation relies on four vectors: F denotes the frontier nodes, MD is the minimum distance for each node, MDT the temporary MD vector, and P the path to the node (which stores the previous node with the shortest path). In the beginning, only the start node is marked as active on F and its distance on MD is set to 0. All other distances on MD are set to ∞. Each iteration executes two kernels. In the first one, each thread is assigned to a frontier node and it relaxes the distance of all adjacent nodes, updating the MDT vector. This operation uses the atomic function *atomicMin()* to avoid concurrency effects. The second kernel processes all nodes whose distance MD differs from MDT, setting the nodes as new frontiers and updating the MD vector. Finally, P stores the selected path. The algorithm ends when all nodes on F are marked as inactive.

R-Kleene extends the original Kleene algorithm for transitive closure to compute the shortest path from all to all nodes [21]. It performs standard matrix operations changing its internal operators: multiplication with a sum, and sum with min; defining the new operators $\otimes$ and $\oplus$ respectively. The operation with matrices A, B, C that $C = A \otimes B$ is given by $C_{ij} = \min_{k=0}^{n-1} A_{ik} + B_{kj}$. We extend an algorithm already adapted to GPU [22] by adding the features needed to deal with any DC or VI topology, and calculating custom distances on-the-fly. As it uses a recursive structure that is not suitable to GPU and the time-consuming routines are the execution of the new operators, the main recursive flow continues to run on CPU, and only the custom operations are translated to GPU kernels. The operations are implemented based on matrix multiplication using shared memory. When the input matrix reaches a small value (*e.g.*, 16×16) it is calculated on GPU using the Floyd Warshall algorithm.

3.4. Data Clustering Algorithms

K-Means is an unsupervised learning algorithm that can cluster a dataset into k arbitrary clusters using a custom similarity function [23]. The algorithm selects k random data points as cluster centers, and iteratively, attributes each data point to a cluster that is similar or near to its center, and then recalculates the center of each cluster. Determining k in a graph is often difficult as K-Means uses random numbers and cluster quality can be affected by bad generators. We define the similarity function of a node as the sum of all distances from it to its cluster's member nodes. There are 3 time-consuming operations in the K-Means implementation for GPU. The first is choosing to which cluster each node belongs; an operation achieved by a kernel that assigns each thread to a node, and computes its distance to all clusters centers, setting its cluster to the nearest one. After that, all centers are recalculated and a kernel computes the distance of each node to other nodes in the same cluster. The last step is to decide which node is the center of its cluster, performed with a simple reduction procedure. The algorithm stops when the centers do not change after an iteration.

Markov Clustering (MCL) is an algorithm specific to graphs based on edge's flow and Markov chain. MCL can arbitrarily find the number of clusters in a graph based on its flow, but it contains other parameters that are difficult to configure. It does not have high-level operations to describe node insertion and deletion in a cluster, but at the end of the process, these clusters can be derived from a probability flow matrix. Initially, MCL needs a stochastic flow matrix M constructed using an arbitrary distance function. In the VI allocation, the matrix can be customized using topology attributes like latency or bandwidth. The rest of the algorithm applies two operations at each iteration: the expansion, that is the power p of the stochastic matrix, M^p; and the inflation operator given by $(\Gamma_r(M))_{ij} = \frac{(M_{ij})^r}{\sum_{k=1}^{m}(M_{kj})^r}$ that uses an r to control the inflation power.

The GPU implementation creates a specific kernel for each main operation. For the expansion operation, we use the cuBLAS library with its SEGMM function [24]. We split the inflation operation into 2 kernels as the sum of the column is reused at all cells belonging to the column. The first inflation kernel performs a basic reduction sum for each column, and the second computes the inflation equation for each cell. Both kernels assign the threads to each matrix column and iterate over the cells. The algorithm stops when the maximum cell difference of the matrix between two iterations is less than an error ϵ. In the end, the residual values at a column j of the matrix M contain the group to which j is assigned. When a node is placed at two groups we select the group with the more significant value.

3.5. Graph Allocation Algorithm

Graph allocation, an important part of the strategies described later, is performed in two steps. The first step iterates over the VI nodes, matching each one to the most appropriated DC node using either BF or WF. If a node cannot be allocated, the algorithm aborts and the VI is

rejected. The second step maps the edges of the VI to the DC paths finding the shortest valid path.

3.6. Speedup Analysis Using Microbenchmarks

The allocation algorithms refactored for GPUs are individually evaluated to compare their speedup against their CPU counterparts. Two GPUs are used for performance analysis (NVIDIA GeForce GTX 1080 /8GB, and NVIDIA Titan XP /12GB), hosted by a machine Intel i7 2600K / 32GB RAM. The machine runs Ubuntu 17.04 Server with CUDA 9.0.176, NVIDIA driver 384.81, and GCC 5. We pick the largest DC configurations supported by the GPU to represent the worst-case microbenchmarks. The upper-bound memory limit is given by MCL and R-Kleene ($O(n^2)$). As the GTX 1080 board has 8GB of memory, we use a Fat-Tree $k = 48$ (30528 nodes and 165888 edges), BCube $7, 4$ (28812 nodes and 84035 edges), and DCell $11, 2$ (19152 nodes and 35112 edges), with a single precision variable type. Topology sizes are larger than those identified in the literature. The results are public available [25] and the microbenchmarks highlight that GPU speed up most algorithms for VI allocation. It is worth pointing out that multiple executions of an algorithm, or combinations of algorithms, may be required to find a suitable allocation.

4. GPU-Driven Allocation Framework

4.1. Framework Structure

The framework was developed as a header-only C++ library based on templates. Moreover, the core data structure, a graph, can be created with custom variables. The elementary composing functions can be extended through user-specified configurations (number of parameters and types of variables) for comparing and processing variables (nodes and edges weights). No prior knowledge on GPU programming is required to use the framework since processing is automatically parallelized by the execution engine. The graph structure requires 2 template arguments, the variable type (integer, float, or double) and the graph's variable struct (containing node capacity, CPUs, memory) for the topology. The framework has pre-defined types to represent weights: CPU and memory for nodes and bandwidth for the links. Moreover, different functions require their own methods that are customized and placed along with structures.

4.2. Allocation Strategies

We propose 4 allocation strategies (Figure 1) for combining the GPU algorithms for data clustering, vertex and edge comparison, and graph allocation. **Path Oblivious Allocation** consists in executing either WF or BF for node comparison while ignoring shortest path selection, whereas **Graph Allocation**, §3.5, executes either WF or BF for node comparison followed by Dijkstra. The algorithm used for **Clustering** depends on the network topology; MCL is used for Fat-Tree and K-Means for BCube and DCell. Under each strategy, when an allocation is not viable, the VI is rejected and the previous DC state is restored.

Strategy 1 is the direct application of the graph allocation algorithm with BF or WF for comparing nodes

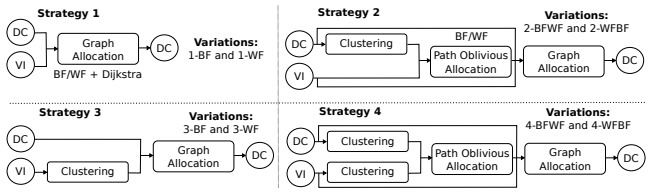

Figure 1. GPU-based allocation strategies.

followed by Dijkstra for path selection, hence resulting in the variations 1-BF and 1-WF.

Strategy 2 uses the data clustering algorithm (§3.4) in the DC graph for reducing the number of comparisons need for finding a solution. The strategy consists of 2 steps: *(i)* mapping the VI to DC clusters performed in a path oblivious manner, and then *(ii)* mapping and extending the small DC graph on the original graph. The extension is the inverse operation of clustering. The variations consist of using BF first and WF second for the allocation steps or vice-versa.

Strategy 3 uses the data clustering algorithm on the VI graph. Different from Strategy 2, it uses only the graph allocation step as only the VI is clustered. The mapping must set the clustered VI to a unique DC node. Then the strategy embeds and extends the VI graph on the original one. The quality of the clusters is crucial because the allocation can be impossible if clusters are too large. Again, there are two possible variations for this using either BF or WF approach.

Strategy 4 applies the data clustering algorithm to both graphs. Similar to Strategy 2 it has two allocation steps. First, the strategy allocates a clustered VI graph into a clustered DC graph. Second, the strategy expands the allocation to the original DC topology, whereby mapping the original VI respecting the clusters discovered on step one. The variations of this strategy also consist of using BF first and WF second or vice-versa.

5. Experimental Setup & Analysis

A discrete event simulator was included in the framework to drive the arrival and termination of requests. The simulation spans the submission of 3000 VIs to be allocated on a cloud DC distributed over 100 discrete intervals. A VI can last from 10 to 50 intervals. All random variables are drawn from uniform distributions, and the results are means of 10 executions varying the simulation seed.

Five metrics were selected to evaluate the simulation: ***(i)*** **Acceptance rate:** the percentage of accepted requests. ***(ii)*** **Number of allocated vCPUs:** provides insights on the use of computing DC resources. ***(iii)*** **Fragmentation:** the percentage of physical resources hosting at least one request. ***(iv)*** **DC footprint:** a metric used to support load balancing algorithms, it measures the number of active physical resources in a DC. ***(v)*** **Execution time:** essential to showing the potential application of GPU to supporting VIs allocation on large-scale cloud DC.

5.1. VI Requests and Cloud DC

VIs have a traditional tree structure formed by 4 levels, each level having from 4 to 7 times more nodes than the upper level. The largest request has 400 nodes, being 343

VMs and 57 virtual switches, whereas the smallest has 85 nodes, with 64 VMs and 21 virtual switches. Regarding QoS requirements, VMs request minimum CPU and memory configuration whereas network links request minimum bandwidth. The VMs composing a VI are homogeneously defined based on *T2.medium* and *T2.xlarge* Amazon EC2 instance types. The virtual link bandwidth is randomly selected from two classes, the CPU-intensive applications (3 Mbps) and the network-intensive (30 Mbps).

The configuration for DC servers is based on the Amazon hardware to approximate the simulation from real cloud configuration. Servers have 20 CPUs and 256GB RAM. The DC links have bandwidth based on the network topology (§2.1). The Fat-Tree has 10 Gbps for the connection between core and aggregation switches and 1 Gbps for others links. In the BCube and DCell topology, all links have 1 Gbps. The simulation is executed with 3 large-scale DCs with similar numbers of servers: Fat-Tree 40 (16000 servers), DCell 11, 2 (17556 servers) and BCube 7, 4 (16807 servers). The configuration for each topology was defined to accommodate all data (DC, requests and algorithms structures) on GPU memory. As the strategies under analysis combine the individually benchmarked algorithms (§3.6), the maximum number of DC nodes is slightly reduced, but it still extrapolates the scenarios considered in the literature.

5.2. Experimental Results

VI Acceptance Rate and Number of vCPUs. Figure 2 summarizes the results on acceptance rate and number of vCPUs allocated for Fat-Tree, BCube, and DCell topologies.

Figure 2. Acceptance rate and number of allocated vCPUs.

The top charts show that 2-BFWF and 4-BFWF are the only strategies with less than 60% of acceptance rate. 1-WF gives the best acceptance rate, around 80%, for all topologies. BCube is the most stable topology with acceptance rate between 60% and 80% for all algorithms. The bottom charts show the number of allocated vCPUs, and we observe that low acceptance rate results in few allocated vCPUs, *i.e.*, FatTree and DCell with 2-BFWF algorithm.

The results show that 2-BFWF, 3-BF, 3-WF and 4-BFWF accept less VI requests compared to 1-BF, 1-WF, 2-WFBF and 4-WFBF strategies. There are two main reasons for this. In Strategy 3, although clustering can decrease the number of comparisons to find a solution, the resulting cluster size can request more capacity than the residual on DC nodes, hence constraining the allocation. Under BFWF variants, the strategies execute the first embedding step (§4.2), but the later expansion and mapping processes. This also occurs in the WFBF variants, but only when the DC is nearly full. For that reason, we select the variants 1-BF, 1WF, 2WFBF and 4-WFBF to continue the analysis.

DC Footprint and Fragmentation. The footprint and fragmentation for all topologies are depicted in Figure 3.

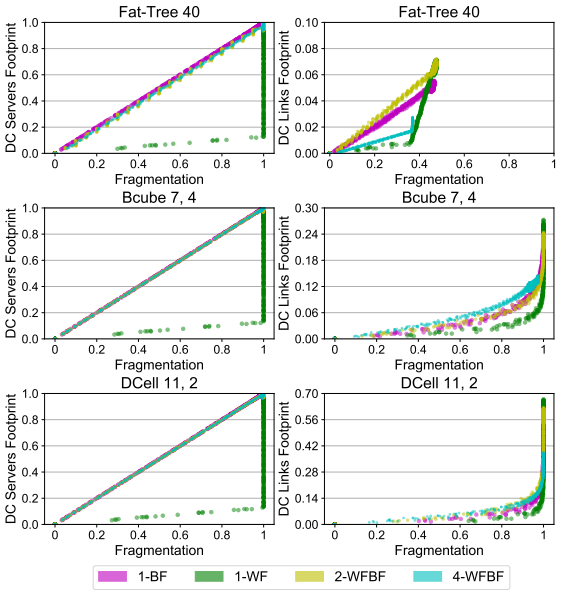

Figure 3. DC footprint and fragmentation for servers and links.

This comparison permits the analysis on how well the resources are consumed, since high fragmentation with low footprint means that many servers and links are online with low utilization, hence increasing DC cost. However, the same growth rate of fragmentation and footprint means that the resources are being fully utilized before turning on another equipment, decreasing the DC cost but increasing the chance of failure. 1-WF differs from other strategies as it reaches 100% host fragmentation between 15% and 20% footprint, which is caused by its WF police, spreading out the VI across the DC. The other variants have similar results, where the host footprint and fragmentation grow together linearly until 100%. These results indicate that 1-WF is suitable to cases where the scattering is necessary to avoid faults. The variants 1-BF, 2-WFBF, and 4-WFBF accommodate most VMs near one another, which can save bandwidth, but can increase the chance that an eventual fault strikes multiple VMs. The behavior on link fragmentation and footprint is different. For Fat-Tree, link fragmentation and footprint are less than 50% and 8%, respectively, for all variants. Variant 4-WFBF has an interesting behavior where it has fewer link utilization (lower footprint), and when it reaches a certain fragmentation ($\pm 40\%$) it only starts to increase footprint with lower fragmentation variation. For two other topologies, link fragmentation reaches 100% when footprint is 30% (BCube) and 70% (DCell) and three variants have similar results. 1-WF, 1-BF, and 2-WFBF reach higher link utilization than 4-WFBF.

Network Load. Figure 4 depicts the DC link utilization in percent categories. Strategy 4 has consistent characteris-

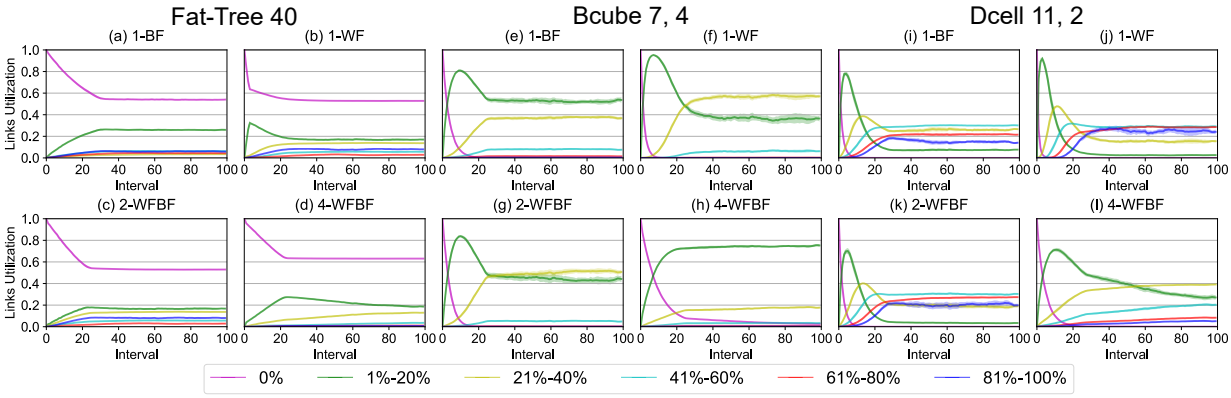

Figure 4. DC links load evolution for the 3 topologies: *(a), (b), (c),* and *(d)* for Fat-Tree with $k = 40$; *(e), (f), (g),* and *(h)* for BCube $7, 4$; and *(i), (j), (k),* and *(l)* for DCell $11, 2$. Results are presented as ranges to identify the average load during the analysis.

tics across all topologies. First, it decreases the use of links as the number of edges with 0% utilization demonstrates. Second, it avoids overloading the links. For example, the number of edges whose utilization is greater than 81% is reduced on the Fat-Tree and DCell topologies when comparing Strategy 4 to the other three. For the BCube topology, however, it results in $1\% - 20\%$ utilization rather than $21\% - 40\%$; different from the other strategies.

Execution Time. The time for allocating VI requests with GPU-accelerated algorithms is presented in Figure 5.

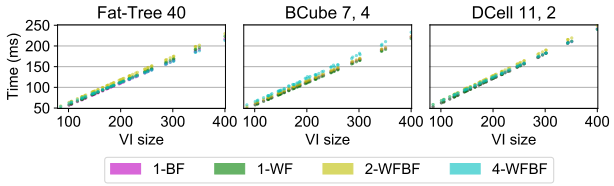

Figure 5. VI allocation time in milliseconds. The time grows linearly with the VI size, independent of the strategy or DC size.

The time grows linearly with the VI size, independent of the strategy. Strategies that use clustering have higher execution time due to the execution of MCL or K-Means on DC topologies. However, this step is only required at the beginning of the execution (it is executed once for allocating 3000 requests). These times are intentionally not showed in Figure 5. In summary, the average time is similar to all strategies, regardless the DC topology.

Key Observations. Strategy 1-BF has lower acceptance rate but nearly the identical load. This scenario can be explained as strategy 1-BF allocates more requests composed of *T2.xlarge* type (with double the node capacity requirements), while the other 3 strategies allocate more VIs based on *T2.medium* types. For fragmentation, variant 1-BF has lower fragmentation in nodes, whereas the same behavior is not observed for links. Strategy 4-WFBF is the opposite, having the lowest edge fragmentation, while strategy 1-WF has the highest fragmentation on both cases, and 2-WFBF comes second in both cases.

An individual discussion is required for clustering algorithms. Although MCL and K-Means present similar results, the definition of their arguments is a challenging task and can change for each DC topology. For instance, K-Means was configured with 70 groups for B-Cube and DCell. We found that value by dividing the number of servers by the average request size. Also, we selected MCL for the Fat-Tree because K-Means yielded poor clusters on it.

Regarding to execution time, the GPU-accelerated algorithms computed solutions for large-scale DC topologies in within a few milliseconds, pointing out to an applicability to real public clouds. Finally, it is worth noting that the analyzed strategies are an example of how the GPU-accelerated algorithms can be combined to compose VI allocation algorithms. Most consolidation algorithms from the literature (§2.2) can be refactored and expressed by the framework. The main goal of the present analysis was to demonstrate the applicability of the framework, without arguing for the application of any specific algorithm.

6. Considerations

The allocation of cloud DC resources for hosting VIs is a challenging problem for both public and private cloud providers. In this paper, we created a GPU accelerated framework for allocating resources to VI. The algorithms enable a series of customizations for designing new allocation methods, without requiring a user to know how to program GPUs. By using microbenchmarks, all algorithms were evaluated and their execution time compared against CPU-based counterparts. In addition to achieving good speedup, the framework allows for considering problem sizes much larger than those found in the literature. To demonstrate the use of the framework, four allocation strategies were introduced and evaluated for allocating VI requests atop DC topologies. Finally, the work extended the existing literature proposing the application of GPUs to support the allocation of VI requests on large-scale DCs. Although the focus of this work is on VIs allocating, similar placement problems emerge in other areas such as component-based applications and distributed data stream processing systems [26].

Acknowledgments. This research was carried out at the LabP2D, and supported by NVIDIA, FAPESC, and UDESC.

References

[1] A. Fischer *et al.*, "Virtual network embedding: A survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 4, pp. 1888–1906, Fourth 2013.

[2] E. Amaldi *et al.*, "On the computational complexity of the virtual network embedding problem," *Electronic Notes in Discrete Mathematics*, vol. 52, pp. 213–220, 2016.

[3] M. Al-Fares *et al.*, "A scalable, commodity data center network architecture," *SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, Aug. 2008.

[4] C. Guo *et al.*, "Bcube: A high performance, server-centric network architecture for modular data centers," in *Proc. of the SIGCOMM Conf. on Data Communication*. NY, USA: ACM, 2009, pp. 63–74.

[5] ——, "Dcell: a scalable and fault-tolerant network structure for data centers," in *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4. New York, NY, USA: ACM, 2008, pp. 75–86.

[6] S. Cook, *CUDA Programming: A Developer's Guide to Parallel Computing with GPUs*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2013.

[7] L. Page *et al.*, "The pagerank citation ranking: Bringing order to the web." *Stanford InfoLab, Technical Report*, no. 1999-66, Nov. 1999.

[8] M. Rost *et al.*, "Beyond the stars: Revisiting virtual cluster embeddings," *SIGCOMM Computer Communication Review*, vol. 45, no. 3, pp. 12–18, Jul. 2015.

[9] F. R. de Souza *et al.*, "QoS-Aware Virtual Infrastructures Allocation on SDN-based Clouds," in *Proc. of the Int. Symp. on Cluster, Cloud and Grid Computing (CCGrid)*. Madrid, Spain: IEEE/ACM, 2017.

[10] R. Niranjan Mysore *et al.*, "Portland: A scalable fault-tolerant layer 2 data center network fabric," in *Proc. of the SIGCOMM Conf. on Data Communication*. Barcelona, Spain: ACM, 2009, pp. 39–50.

[11] A. Singh *et al.*, "Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, Aug. 2015.

[12] M. Kliegl *et al.*, "The generalized DCell network structures and their graph properties," *Microsoft Research, Technical Report*, 2009.

[13] N. M. M. K. Chowdhury *et al.*, "Virtual network embedding with coordinated node and link mapping," in *IEEE INFOCOM 2009*. Rio de Janeiro, RJ, Brasil: IEEE, April 2009, pp. 783–791.

[14] R. de Oliveira and G. P. Koslovski, "A tree-based algorithm for virtual infrastructure allocation with joint virtual machine and network requirements," *International Journal of Network Management*, vol. 27, no. 1, pp. e1958–n/a, 2017, e1958 nem.1958.

[15] N. Farooq Butt *et al.*, *Topology-Awareness and Reoptimization Mechanism for Virtual Network Embedding*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 27–39.

[16] R. N. Calheiros *et al.*, "CloudSim: A toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms," *Softw. Pract. Exper.*, vol. 41, no. 1, pp. 23–50, Jan. 2011.

[17] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed., ser. EngineeringPro collection. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics (SIAM), 2003.

[18] N. T. Duong *et al.*, "Parallel pagerank computation using GPUs," in *Proc. of the Third Symp. on Information and Communication Technology*. NY, USA: ACM, 2012, pp. 223–230.

[19] M. Yu *et al.*, "Rethinking virtual network embedding: Substrate support for path splitting and migration," *SIGCOMM Computer Communication Review*, vol. 38, no. 2, pp. 17–29, Mar. 2008.

[20] P. Harish *et al.*, "Large graph algorithms for massively multithreaded architectures," *International Institute of Information Technology, Hyderabad, Technical Report*, 2009.

[21] P. D'Alberto and A. Nicolau, "R-kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," *Algorithmica*, vol. 47, no. 2, pp. 203–213, 2007.

[22] A. Buluç *et al.*, "Solving path problems on the gpu," *Parallel Comput.*, vol. 36, no. 5-6, pp. 241–253, Jun. 2010.

[23] S. Marsland, *Machine learning: an algorithmic perspective*, 2nd ed. Boca Raton, FL: Chapman and Hall/CRC Press, T&F Group, 2015.

[24] NVIDIA, *CUDA C Programming Guide, NVIDIA Documentation*. Santa Clara, CA, USA: NVIDIA Corporation, 2017.

[25] L. L. Nesi *et al.*, "GPU-accelerated algorithms for allocating virtual infrastructure in cloud data centers," in *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2018.

[26] V. Cardellini *et al.*, "Optimal operator placement for distributed stream processing applications," in *10th ACM DEBS '16*. New York, USA: ACM, 2016, pp. 69–80.