The Cloud as an OpenMP Offloading Device

Hervé Yviquel, Guido Araújo

The Cloud as an OpenMP Offloading Device

Hervé Yviquel and Guido Araújo
Institute of Computing
University of Campinas (UNICAMP)
Campinas, Brazil
{herve.yviquel,guido}@ic.unicamp.br

***Abstract*—Computation offloading is a programming model in which program fragments (e.g. hot loops) are annotated so that their execution is performed in dedicated hardware or accelerator devices. Although offloading has been extensively used to move computation to GPUs, through directive-based annotation standards like OpenMP, offloading computation to very large computer clusters can become a complex and cumbersome task. It typically requires mixing programming models (e.g. OpenMP and MPI) and languages (e.g. C/C++ and Scala), dealing with various access control mechanisms from different clouds (e.g. AWS and Azure), and integrating all this into a single application. This paper introduces the cloud as a computation offloading device. It integrates OpenMP directives, cloud based map-reduce Spark nodes and remote communication management such that the cloud appears to the programmer as yet another device available in its local computer. Experiments using LLVM, OpenMP 4.5 and Amazon EC2 show the viability of the proposed approach and enable a thorough analysis of the performance and costs involved in cloud offloading. The results show that although data transfers can impose overheads, cloud offloading can still achieve promising speedups of up to 86x in 256 cores for the *2MM* benchmark using 1GB matrices.**

I. Introduction

Parallelizing loops is a well-known research problem that has been extensively studied. Most of the approaches to this problem use DOALL [1], DOACROSS [2], DSWP [3], vectorization [4], data rearrangement [5] and algebraic and loop transformations [6] to improve program performance. On the other hand, the combination of large data-center clusters and Map Reduce based techniques, like those found in Internet search engines [7], has opened up opportunities for cloud-based parallelization, which could eventually improve program performance.

Although there is a number of approaches to loop parallelization, in general-purpose computers this is typically achieved through message passing programming models like MPI [8] or multi-threading based techniques such as those found in the OpenMP standard [9]. OpenMP is a directive-based programming model in which program fragments (e.g. hot loops) are annotated to ease the task of parallelizing code. The last version of the OpenMP standard [10] (Release 4.5) introduces new directives that enable the transfer of computation to heterogeneous computing devices (e.g. GPUs). From the programmer viewpoint, a program starts running on a typical processor host, and when an OpenMP annotated code fragment is reached, the code is transferred to the indicated device for execution, returning the control flow to the host after completion, a technique called *offloading*.

Although OpenMP offloading has been extensively used in combination with powerful computing devices like GPUs, offloading computation to very large clusters can become a complex and cumbersome task. It typically requires mixing different programming models (e.g. OpenMP and MPI) and languages (e.g. C/C++ and Scala), dealing with access control mechanisms from distinct cloud services, while integrating all this together into a single application. This task can become a major programming endeavor that can exclude programmers who are not parallel programming experts from using the huge computational resources available in the cloud [11].

To address such problem, this paper integrates OpenMP directives, cloud based map-reduce Spark and remote communication management into a single OpenMP offloading device which can be seen by the programmer as available in its local computer. To achieve that, it relies on the OpenMP accelerator model to include the cloud as a new device target. The cloud resources (e.g. execution nodes, data storage) are identified using a specific configuration file and a runtime library allows the programmer to get rid of all glue code required for the interaction with the cloud infrastructure. The main contributions of this paper are the following:

- It introduces the cloud as an OpenMP computation device making the task of mapping local source code to the cloud transparent to the programmer, a useful tool specifically for those programmers who do not master parallel programming skills and cloud computing;
- It proposes a new cloud-based parallel programming model which combines traditional parallelization techniques with map-reduce based computation to enable the generation of parallel distributed code;

The remainder of this paper is organized as follows. First, we start by introducing the basic concepts involved in directive based programing and cloud computing (Section II). We then describe the proposed approach in Section III. Section IV presents and analyzes the experimental results, and Section V discusses the related works. Finally, we conclude in Section VI.

II. Background

Cloud computing has been considered a promising platform which could free users from the need to buy and maintain expensive computer clusters, while enabling a flexible and

pay-as-needed computational environment. Although it has been successfully used to handle the rise of social media and multimedia [12], all such systems have been designed using a programming abstraction that clearly separates the input/output of local data from cloud computation. This goes against a clear tendency in computing to easy the integration of data collection to the huge resources available in the cloud, as demanded by modern mobile devices and Internet-of-Things (IoT) networks. For example, by collecting data from a cellphone and transparently sending it to the cloud, one could use expensive Machine Learning (ML) algorithms to identify the best device parameters, thus tunning its operation to the user usage profile. Easing the integration of local code with the cloud is central to enable such type of applications.

The MapReduce [7] programming model associated with the Hadoop Distributed File System (HDFS) [13] has become the *de facto* standard used to solve large problems in the cloud [14], [15]. A generalization of the MapReduce model, Spark [16] has enabled the design of many complex cloud based applications and demonstrated very good performance numbers [17], [18], [19], [20]. To achieve such performance, the Spark runtime relies on an innovative data structure, called *Resilient Distributed Dataset* (RDD) [16], which is used to store distributed data collections with the support of parallel access and fault-tolerance.

```
1  void MatMul(float *A, float *B, float *C) {
2    // Offload code fragment to the cloud
3    #pragma omp target device(CLOUD)
4    #pragma omp map(to: A[:N*N], B[:N*N]) map(
         from: C[:N*N])
5    // Parallelize loop iterations on the cluster
6    #pragma omp parallel for
7    for(int i=0; i < N; ++i)
8      for (int j = 0; j < N; ++j)
9        C[i * N + j] = 0;
10       for (int k = 0; k < N; ++k)
11         C[i * N + j] += A[i * N + k] * B[k * N
               + j];
12   // Resulted matrix 'C' is available locally
13 }
```

Listing 1: Cloud offloading of matrix multiplication using the OpenMP accelerator model

Although it has been used to solve many large-scale problems, the combination of MapReduce and HDFS has not become a programming model capable of turning the cloud into a computing device easily used by non-expert programmers. On the other hand, the emergence of multicore computing platforms has enabled the adoption of *directive-based programming models* which have simplified the task of programming such architectures. In directive-based programming, traditional programming languages are extended with a set of directives (such as *C pragma*) that informs the compiler about the parallelism potential of certain portions of the code, usually loops but also parallel sections and pipeline fragments. OpenACC [21] and OpenMP [9] are two examples of such language extensions which rely on thread-level parallelism. Due to its simplicity and seamless mode, OpenMP is probably the most popular directive-based programming interface in use today.

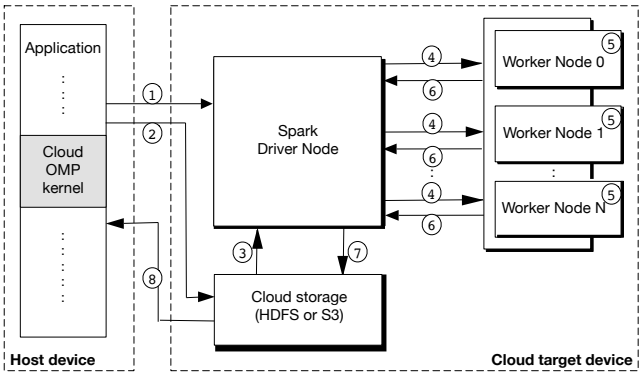


Fig. 1: Workflow of the offloading of a kernel execution to a cloud device

Listing 1 presents a simple program loop describing a matrix multiplication which was annotated with OpenMP directives in order to offload the computation to an accelerator. In the OpenMP abstract accelerator model, the *target* clause defines the portion of the program that will be executed by the target device. The *map* clause details the mapping of the data between the host and the target device: inputs (A and B) are mapped *to* the target, and the output (C) is mapped *from* the target. While typical target devices are DSP cores, NVIDIA GPUs, Xeon Phi accelerators, etc., this paper introduces the cloud as yet another target device available from the local computer, giving the programmer the ability to quickly expand the computational power of its own computer to a large-scale cloud cluster.

III. Cloud offloading infrastructure for OpenMP

Instead of parallelizing program fragments across heterogeneous cores within a single computer, our runtime automatically parallelizes loop iterations by offloading kernels from the local computer across multiple machines of a cloud cluster. To achieve that it uses the Apache Spark framework, while transparently providing desirable features like fault tolerance, data distribution and workload balancing.

As shown in Figure 1, the execution of the local OpenMP annotated code on the remote cloud device has the following workflow. First, the programmer configures the credentials of a Spark cluster previously deployed using its favorite cloud service. The program is then started by the programmer in its own local machine and runs locally until the OpenMP annotated code fragment is reached. A method is then called to initialize the cloud device (1). Offloading is done dynamically, and thus if the cloud is not available the computation is performed locally. The runtime sends the input data required by the kernel as binary files to a cloud storage device (e.g. AWS S3 or any HDFS server) (2). After all the input data has been transmitted, the runtime submits the job to the Spark cluster and blocks until the end of the job execution. The *driver node* which is in charge of managing the cluster reads the

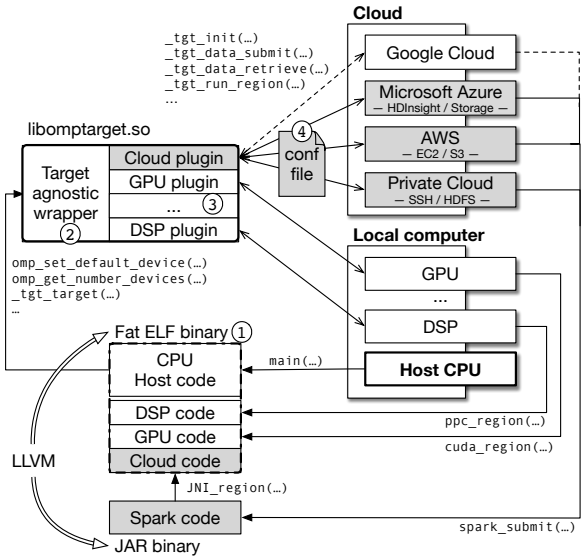


Fig. 2: Modular implementation of the OpenMP accelerator model; in gray is what we implemented to enable the cloud as a novel device

input data from the cloud file system ③, transmits the input data and distributes the loop iterations across the Spark *worker nodes* ④, which are in charge of the computations. Next, the *worker nodes* run the mapping function that compute the loop body in parallel ⑤. The output of the loop is then collected, reconstructed by the driver ⑥ and stored into the cloud storage ⑦ to be transmitted back to the local program ⑧, which then continues the execution on the local machine.

A. Offloading to the cloud cluster

Our workflow relies on a flexible implementation of the OpenMP accelerator model, presented in Figure 2. Such an implementation was developed by Jacob et al. within the OpenMP offloading library [22] (known as *libomptarget*) and the LLVM compiler [23]. Their implementation relies on runtime calls made by the host device for the execution of the offloaded code on the target device. In the initial implementation, offloading was implemented only for typical devices like general-purpose processors (x86 and PowerPC) and NVIDIA GPUs (running CUDA code). In this paper, we extended the LLVM compiler to generate code for Spark-based cloud devices and the *libomptarget* library to allow the offloading of data and code to such devices[1], shown in gray in Figure 2. In order to ease the implementation of new accelerators, Jacob et al. [22] decomposed their implementation in distinct components:

① *Fat binary generated by LLVM* – which contains host and target codes. While host code (contained in the `main(...)` function) and target codes (such as function `ppc_region(...)`) are typically embedded in the same fat binary using the ELF format, our cloud target requires an additional file to be generated: the Scala code describing the Spark job (compiled to JAR binary). When submitting the job to the cluster, the *driver node* runs the Scala program and distributes the loop iteration among the *worker nodes*. Then, the workers natively run (in C/C++) the function describing the loop body (`JNI_region(...)`) through the Java Native Interface (JNI) so as to avoid the translation of C/C++ code to Scala. Obviously, this code had to be compiled to a binary format compatible with the architecture of the cloud processors;

② *Target-agnostic offloading wrapper* – which is responsible for the detection of the available devices, the creation of devices' data environments, the execution of the right offloading function according to the device type. The wrapper implements a set of user-level runtime routines (such as `omp_get_num_devices(...)`) and compiler-level runtime routines (such as `_tgt_target(...)`) which allow the host code to be independent of the target device type;

③ *Target-specific offloading plug-ins* – which performs the direct interaction with the devices, according to their architecture and provides services such as the initialization and transmission of input and output data, and the execution of offloaded computation. In our case, the cloud-specific plugin is used to initialize the cluster, to compress and transmit the offloaded data through the cloud file storage (HDFS or S3), and to submit the Spark jobs through SSH connection.

There are some major differences when using the cloud to offload computation when compared to other traditional target devices, such as GPUs. For instance, the host-target communication overhead might be reduced by compressing offloaded data, and transmitting them in parallel. Our cloud plugin automatically creates a new thread for transmitting each offloaded data (possibly after gzip compression if the data size is larger than a predefined minimal compression size). Additionally, the user can choose to print the log messages of Spark to the standard output of the host computer to check the current state of the computation. Another remarkable difference is that cloud devices cannot be detected automatically since they are not physically hosted at the local computer. As a matter of fact, the user has to provide an identification/authentication information (e.g. login) to allow the connection of the current application to the cloud service which will be used for offloading. Our cloud plugin reads at runtime a configuration file ④ to properly set up the cloud device and to avoid the need to recompile the binary (assuming compatible instruction-sets). Besides the login information, the configuration file also contains the address of the Spark driver as well as the address of the cloud file storage.

To allow an easy portability over existing cloud services, our cloud plugin was implemented as a modular infrastructure where the communication with the cloud can be customized for each existing cloud service by taking into account their specificities (e.g. storage services, security mechanisms, etc.).

[1]Our cloud offloading workflow is implemented within an open-source toolset which is freely available at http://ompcloud.github.io

For now, our plugin supports computation offloading to Spark clusters running within a private cloud, Amazon Elastic Compute Cloud (EC2), or Microsoft Azure HDInsight. We also support data offloading to HDFS, Amazon Simple Storage Service (S3) and Microsoft Azure Storage. This approach can be easily extended to support other commercial cloud services like Cloudera, or Google Cloud. Moreover, during offloading our library is also able to (on-the-fly) start and stop virtual machines from the EC2 service. In other words, the EC2 instance can be started when offloading the code and stopped after it ends its execution. As a result, the programmer can automatically control the usage of the cloud infrastructure, thus allowing him/her to pay for just the amount of computational resources used.

B. Extending OpenMP for distributed data partitioning

One central issue in distributed parallel execution models is to enable a data partitioning mechanism that assigns a specific data block to the worker node containing the kernel code that will use it. By doing so, programs can considerably benefit from locality, thus reducing the overhead of moving data around interconnecting networks. Nevertheless, automatically data partitioning is a hard task which cannot typically be achieved solely by the compiler or runtime. In most applications the programmer knowledge is essential to enable an efficient data allocation. Unfortunately, the OpenMP standard does not have directives specifically designed for data partitioning within offloaded regions. As a result, the programming model proposed in this paper extends the use of the OpenMP directive *target data map* so as to allow the programmer to express the data distribution to the cloud Spark nodes. No syntax modification was required in this directive; it was only used in a way which is mentioned as *undefined behavior* by the current OpenMP specification. To do that, the programmer should indicate after the *to/from* specifier of the *map* directive the first element of the partitioned data block followed by colon and the last element of the corresponding block.

```
1  #pragma omp target device(CLOUD)
2  #pragma omp map(to: A[:N*N], B[:N*N]) map(from:
       C[:N*N])
3  #pragma omp parallel for
4  for(int i=0; i < N; ++i)
5  #pragma omp target data map(to: A[i*N:(i+1)*N])
       map(from: C[i*N:(i+1)*N])
6    for (int j = 0; j < N; ++j)
7      C[i * N + j] = 0;
8      for (int k = 0; k < N; ++k)
9        C[i * N + j] += A[i * N + k] * B[k * N +
             j];
```

Listing 2: Extending OpenMP data map directive to enable dynamic data partitioning

Consider, for example, the code fragment in Listing 2 extracted from the matrix multiplication example in Listing 1. It is well-known that matrix multiplication $C = A \times B$ implies in multiplying the rows of A by the columns of B storing the result as elements of C. Hence, in order to improve locality the programmer can insert line 5 of Listing 2 to specify the partitioning of the matrices during the iterations of the parallel loop. For example, in line 5 of Listing 2 the rows of matrix A are indexed using variable `i`[2]. By using `map(to:i*N:(i+1)*N)` the programmer states that all elements of row `i` of A range from index `i*N` to `(i+1)*N` and should be allocated into the same Spark node. Please notice that B is deliberately not partitioned because its partition interval depends on the internal loop counter j (indexing the column). In our implementation, the partitioning which reduces the amount of data moved on the network is performed by the Spark driver node, but Spark only knows the values taken by the loop counter of the outer loop (i.e. the parallel for). Partitioning B would require to coalesce the internal loop into the external one which would increase the number of iterations, thus reducing the granularity of the parallelization, therefore increasing scheduling and communication overhead. For this reason, each worker node will receive a full copy of B to perform its part of the computation. In reality the communication overhead will be limited by the efficiency of BitTorrent protocol used by Spark to broadcast variables.

C. Matching Spark execution model

As said before and shown in Figure 2, Spark clusters are composed by one driver node associated with a set of worker nodes (or simply *workers*). The driver is in charge of communication with the outside world (i.e. host computer), resource allocation and task scheduling. The workers perform computation by applying operations, mostly *map* and *reduce*, in parallel on large datasets. In our programming model, given that the loop has been annotated using a *parallel for* clause, the programmer assumes that it is a DOALL loop. Thus, the different iterations can be distributed and computed in parallel among cloud cores without any restriction, as no loop-carried dependence exists between iterations.

To achieve such a parallelism, Spark relies on a specific data structure called *Resilient Distributed Dataset* (RDD) [16]. An RDD is basically a collection of data that is partitioned among the workers which apply parallel operations to them. In order to allow the parallel execution of a DOALL loop with N iterations, we build an initial RDD such that:

$$RDD_{IN} = \bigcup_{i=0}^{N-1} \{i, V_{IN}(i)\} \quad (1)$$

$$V_{IN}(i) = \{V_{IN_0}(i), ..., V_{IN_{K-1}}(i)\} \quad (2)$$

where $i \in \{0, ..., N\text{-}1\}$ are the values taken by the loop index during loop execution and $V_{IN}(i)$ is the set of K input variables (i.e. *r-values*) read during the execution of the loop body at iteration i. For any iteration i, each input $V_{IN_k}(i)$ with $k \in \{0, ..., K\text{-}1\}$ can be either a full variable or a portion of it depending if the programmer has described the partitioning of the variable as presented in Listing 2.

RDD_{IN} is divided automatically by the driver in equal parts and distributed among the workers $w \in \{0, 1, ..., W\text{-}1\}$

[2]Matrices A, B and C in Listing 2 are represented in their linearized forms.

such that:

$$RDD_{IN}(w) = \bigcup_{m=w*\lfloor N/W \rfloor}^{(w+1)*\lfloor N/W \rfloor-1} \{m, V_{IN}(m)\} \quad (3)$$

A map operation (Eq. 4) is then applied to the RDD that passes the values taken by the loop index and the input variables through a function describing the loop body (Eq. 5) and returns a new RDD_{OUT} such that:

$$RDD_{OUT} = MAP(RDD_{IN}, loopbody) \quad (4)$$

$$V_{OUT}(i) = loopbody(i, V_{IN}(i)) \quad (5)$$

$$V_{OUT}(i) = \{V_{OUT_0}(i), ..., V_{OUT_{L-1}}(i)\} \quad (6)$$

$$RDD_{OUT} = \bigcup_{i=0}^{N-1} \{i, V_{OUT}(i)\} \quad (7)$$

where $V_{OUT}(i)$ (Eq. 6) is the set of L output variables (i.e. *l-values*) produced at iteration i by the workers. In fact, each call to the *loopbody* function produces a partial value of the output variables V_{OUT} since each iteration accesses a different part of the output variables in DOALL loops. Similarly to the input variables, each output $V_{OUT_l}(i)$ with $l \in \{0, ..., L\text{-}1\}$ can be either a full variable or a portion of it depending if the programmer has described the partitioning of the variable. As a result, the *loopbody* function will only partially compute output variables at each call, even if we know they are not partitioned. Thus, we need to reconstruct the complete outputs V_{OUT} from those *partial* output values $V_{OUT}(i)$ as expected by the host computer such that:

$$V_{OUT_l} = \begin{cases} Reconstruct(RDD_{OUT}, l) \\ REDUCE(RDD_{OUT}, l, bitor) \end{cases} \quad (8)$$

$$V'_{OUT_l}(u, v) = bitor(V_{OUT_l}(u), V_{OUT_l}(v)) \quad (9)$$

$$V_{OUT} = \{V_{OUT_0}, ..., V_{OUT_{L-1}}\} \quad (10)$$

where $V'_{OUT_l}(u, v)$ is the partial output value obtained by combining $V_{OUT_l}(u)$ and $V_{OUT_l}(v)$ (Eq. 9). If the loop body has several outputs (Eq. 10), RDD_{OUT} will simply be composed of tuples which are reconstructed separately before being written back in different binary files. Thus, we consider the set of all partial values of each output variable $V_{OUT_l}(i)$ as arrays of bytes. If the variable was partitioned, the driver allocates the full variable and writes each value at the right index. If the programmer has not detailed the partitioning, we simply apply a bitwise-or reduction (Eq. 8) to join them together. Additionally, if one of the outputs has been defined as a reduction variable by the OpenMP clause, Spark just performs the reduction using the predefined function instead of the bitwise-or.

To illustrate this process, let us consider the matrix multiplication $C = A \times B$ presented previously in Listing 1. As shown in Figure 3, the Spark driver node gets the files ① representing the input data from the cloud storage (HDFS or S3), loads them as ByteArray objects. It then generates $Rdd(I)$ which contains the successive values taken by the loop index i $(0, .., 15$ in our case), splits A according to the partitioning

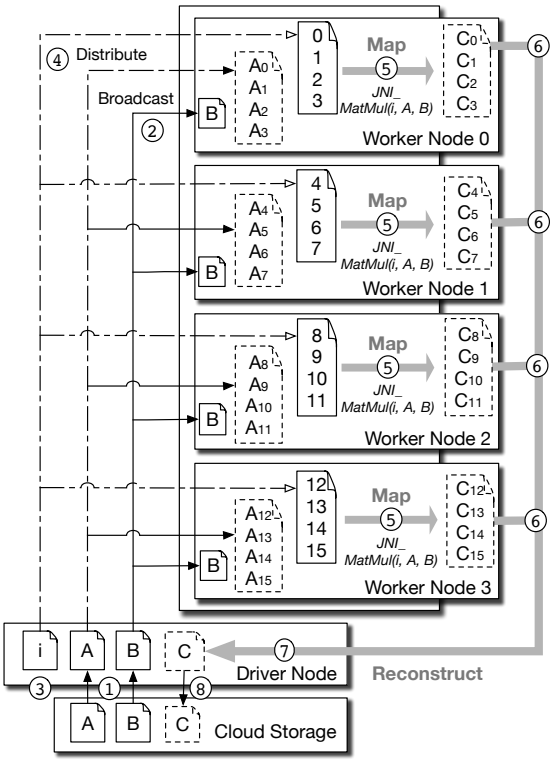


Fig. 3: Using map-reduce computational model to perform matrix multiplication $C = A \times B$ in Spark cluster

bound defined by the user (in function of i), distributes them equitably to the worker cores ④ while broadcasting B ② to all worker nodes. Notice that Spark automatically compresses all data transmitted through the network and use the BitTorrent protocol for broadcasting efficiently. The driver orders a map transformation that applies the $MatMult$ function corresponding to the loop body (through JNI) for each loop iteration using partitions of A, and copies of B. The workers decompress the input data and perform in parallel the mapping tasks assigned to their cores ⑤. As a result, they produce an RDD containing sixteen versions of C which needs to be collected so as to produce the final result for array C. To achieve that the workers compress and send the values to the driver ⑥. Then, the driver starts by allocating variable C to its full size, before sequentially writing at the right index the values contained by each piece of the array C, until obtaining its final format ⑦. The driver finally writes out the final values of C to a new file into the cloud storage ⑧ (line 20), after which the local computer is able to read them back and continue its execution.

In addition, our compiler automatically adjusts the iteration number of the outer-loop according to the cluster size using loop tiling to reduce JNI overhead, such as presented in Algorithm 1 (with C the number of worker cores). Indeed, since each iteration will require one call to JNI, the closer the number of iterations is to the number of cores, the smaller will be the overhead. The total number of cores C is passed as

Algorithm 1 Reducing overhead by tiling the loop to the cluster size

```
// Original parallel for
for i = 0 to N-1 do
    loopbody
end for
```

```
// Transformed for where C is the number of worker cores
for ii = 0 to N-1 by ⌊N/C⌋ do
    for i = ii to min(ii + ⌊N/C⌋ − 1, N-1) do
        loopbody
    end for
end for
```

an argument when Spark is calling the map functions to avoid any recompilation when executing on different clusters. In case some of the input/output variables are partitioned, the lower and upper bounds of the partitions will also be readjusted dynamically according to the tiling size, hence increasing their granularity.

D. Application domain and model limitations

Before moving into the details of the application domain, it is important to highlight that the goal of the programming model proposed herein is to make the resources of the cloud transparently available to the common non-expert programmer who uses a regular laptop and wants to run large workloads. A typical example is a user that locally collects a large amount of data from a scientific experiment, an IoT sensor network or a mobile device and wants to perform some heavy computation on it. It is not a goal of this work to claim a programming model that can deliver HPC type of speedups on a complex specialized scientific application (e.g. ocean simulation): there is already a huge amount of work and programming models (e.g. MPI) that work well in this area [24].

Of course, this specific solution does not match well the computation requirements of any kind of application. First, the problem to be solved has to be sufficiently complex to allow the application to take advantage of the large parallel processing capabilities of the cloud when compared to the overhead cost of the data offloading task. Nevertheless, one might run his application directly from the driver node of the Spark cluster, thus removing the overhead of host-target communication. Second, applications should be described in C/C++ and annotated using directives defined by the OpenMP accelerator model. While this paper presents a matrix multiplication annotated with just one *target* clause, one *target map* and one *parallel for*, our approach also supports more complex OpenMP constructs such as those using several *parallel for* loops within the same *target* region. This is implemented by performing successive map-reduce transformations within the Spark job. Moreover, similar techniques also allow one to implement the offloading of sequential code kernels or nested parallel loops.

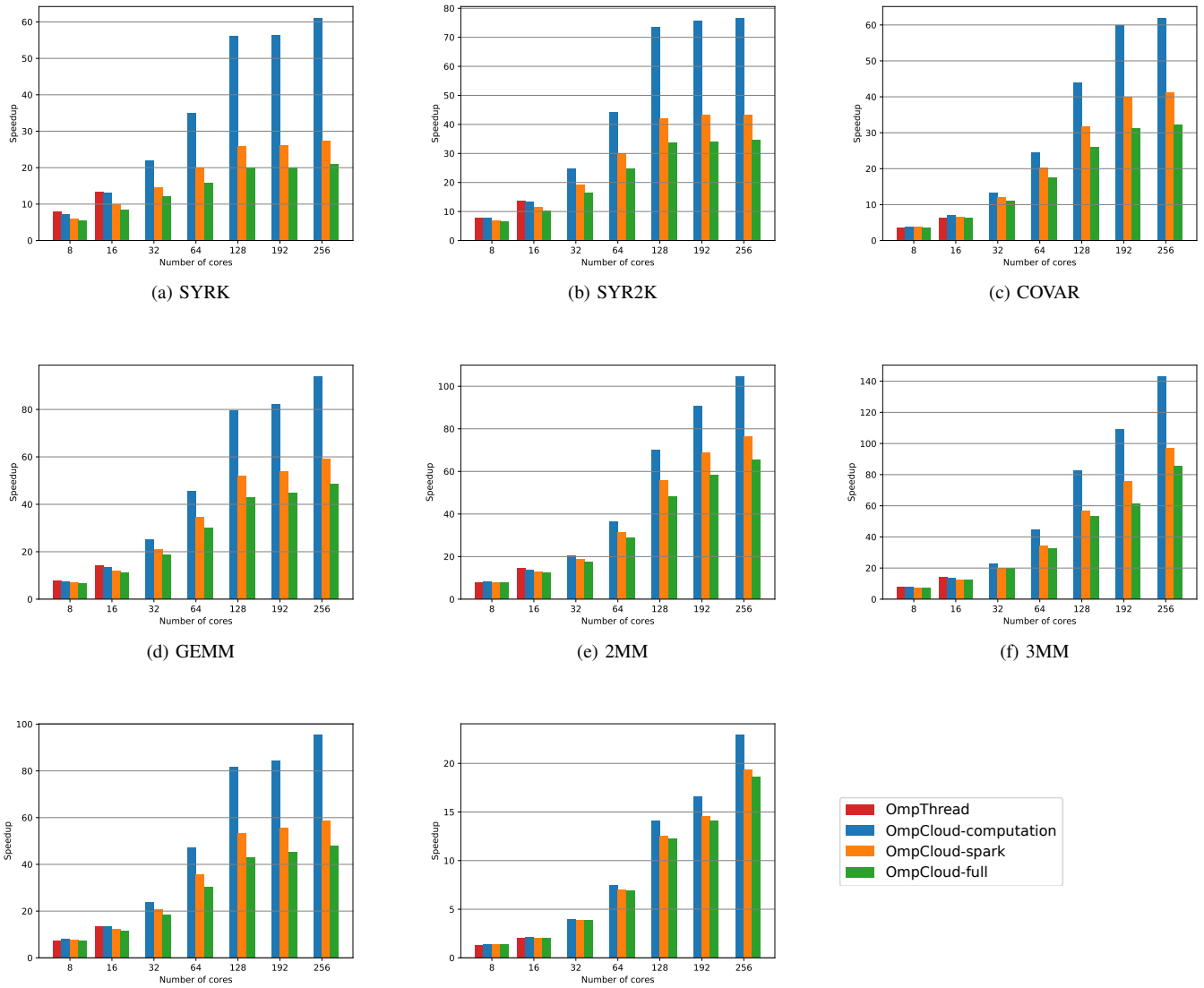
Finally, our cloud device does not support the synchronization constructs of the OpenMP model since Spark relies on a distributed architecture. Thus, offloaded OpenMP regions that use *atomic*, *flush*, *barrier*, *critical*, or *master* directives are not supported. A full implementation of OpenMP on cloud clusters would require a distributed shared memory mechanism [25], [26] which has not yet been proved to be efficient and is incompatible with the map-reduce model. Alternatively, a more restricted programming model suited for distributed nodes could be employed but we believe that the popularity of OpenMP makes it a better choice.

IV. Experimental Results

In our experiments, the local machine is a simple laptop (Intel core i7 and 16GB of RAM) which interacts with an AWS cluster through an Internet connection. Our experiments intend to be a realistic test-case where the client computer is far away from the cloud data-center. The cloud instances were acquired and configured using a third-party script called *cgcloud*[3]. This script allowed us to quickly instantiate a fully operational and highly customizable Spark cluster within AWS infrastructure. For now, the size of the cluster is predefined by the user when running the script but the parallel loop is tiled dynamically to use all instances with the minimal overhead. Our experimental Spark cluster was composed of 1 driver node and 16 worker nodes, all of them running Apache Spark 2.1.0 on top of Ubuntu 14.04. Each node of the cluster is an EC2 instance of type *c3.8xlarge* which have 32 vCPU (executing on Intel Xeon E5-2680 v2) and 60GB of RAM. Each worker is configured to run one Java Virtual Machine (commonly called Spark executor) that manages all 32 vCPUs and a heap size of 40GB. Since each EC2 vCPU corresponds to one hyper-threaded core according to Amazon description (e.g. 1 dedicated CPU core corresponds 2 vCPUs), we configured Spark to assign two vCPUs to each map and reduce task we need to run (*spark.task.cpus=2*). Thus, the following benchmark results are presented according to the number of dedicated CPU cores used by all workers (from 8 to 256 cores which is configured thanks to *spark.cores.max* and *spark.default.parallelism* parameters).

We used benchmarks from the *Polyhedral Benchmark* suite [27] and the *MgBench* [28] suite which were previously adapted for the OpenMP accelerator model. We selected for our experiments the set of benchmarks which contains only the supported OpenMP constructs and which could benefit the most of cloud offloading: *SYRK*, *SYR2K*, *COVAR*, *GEMM*, *2MM* and *3MM* from *Polybench*; and *Mat-mul* and *Collinear-list* from *MgBench*. All data used in the benchmarks consisted of 32-bit floating point numbers (simple precision). The dimension of the datasets used by the benchmarks has been scaled to benefit from the Spark distributed execution model. As an example, most matrices used by the benchmarks have been scaled to about 1GB. Moreover, in order to evaluate the impact of the compression on performance, we have deliberately executed the benchmarks using two types of input data: **sparse** and **dense** matrices. Indeed, sparse matrices are compressed faster with better compression rate.

[3] *cgcloud* is freely available at http://github.com/BD2KGenomics/cgcloud/



(a) SYRK (b) SYR2K (c) COVAR

(d) GEMM (e) 2MM (f) 3MM

(g) Mat-mul (h) Collinear-list

Fig. 4: Average speedup of multicore over single core execution for cloud offloading, and for multi-threaded OpenMP as reference.

Figure 4 presents the execution speedup obtained with the benchmarks parallelized using OpenMP both with traditional multi-threading and cloud offloading (noted *OmpThread* and *OmpCloud* respectively). Cloud-based speedups were computed for the whole offloading time noted *OmpCloud-full* in the Figure, but also detailed for Spark job execution time (without the host-target communication) noted *OmpCloud-spark*, and for the computation time only (i.e. parallel execution of the mapping tasks) noted *OmpCloud-computation*. Only speedups with 8 and 16 threads are presented for *OmpThread* since the largest AWS EC2 instances of type c3 has 16 cores. Comparing to the speedups obtained with *OmpThread* on 8 and 16 cores, the speedup of *OmpCloud* on 8 and 16 cores (i.e. one worker node) revealed small execution time overheads: (a) just 1.8% when considering only *OmpCloud-computation* which confirms the efficiency of JNI to run native code kernels; (b) 8.8% when considering *OmpCloud-spark* which demonstrates the competitive performance of the Spark execution model with respect to multi-threading, even with a driver-worker infrastructure; and (c) 13.6% when considering *OmpCloud-full* which shows the limited cost of offloading data to the cloud even without huge computational power. Globally, all speedups of *OmpCloud* tend to increase with the number of cores (up to 143x/97x/86x respectively with 256 cores for 3MM on Chart 4f). If we compare the result of *OmpCloud-computation* with *OmpCloud-spark* (which includes Spark overhead like intra-cluster communication and task scheduling), the speedups

continue to scale with the number of cores but with an increasing overhead. This is the case of *collinear-list* (Chart 4h) which shows the smaller overheads varying from 0.1% on 8 cores to 15% on 256 cores, or *SYRK* which shows the larger ones varying from 17% to 69%. Interesting enough, for all benchmarks, the host-target communications account for a small share of the total overhead indicating that the most relevant bottleneck comes in fact from one of the phases internal to the cloud cluster, especially when the cluster size increases.

The execution times of cloud offloading are presented in Figure 5. They were measured when running the benchmarks on both sparse and dense matrices to explore the impact of the data type on the performance. Results show that: (a) 2 benchmarks are executed on 8 cores in between 10 and 25min; (b) 5 in between 30min to 1h; and (c) 1 in about 1h30. Although offloading to a larger cluster could probably benefit from even longer execution times, we were limited by the maximal size of the arrays supported by the Java Virtual Machine. As shown in Figure 5, the distribution of the execution time of all benchmarks were broken into 3 parts: *Host-target communication* including compression and transmission overhead between the local computer and the cloud device, *Spark overhead* including scheduling and communication within the cluster, and *computation time* (i.e. the loop iterations executed in parallel through JNI). Such decompositions show that while the computation time decreases as the number of cores increases, the overhead induced by cloud offloading and Spark distributed execution stays constant. Moreover, both overheads increase substantially when processing dense matrices (in comparison with sparse ones) but the variation is negligible for the computation time. This demonstrates that the data type (and especially its compressibility) can have a huge impact on performance because of the host-target and intra-cluster communications. Additionally, results of *collinear-list* presented in Chart 5h show a negligible overhead of the communication and scheduling. In fact, *collinear-list* processes a much smaller amount of data than the other benchmarks, showing that cloud offloading scales well when the dataset size stays small according to the computation (i.e. High computation to communication ratio).

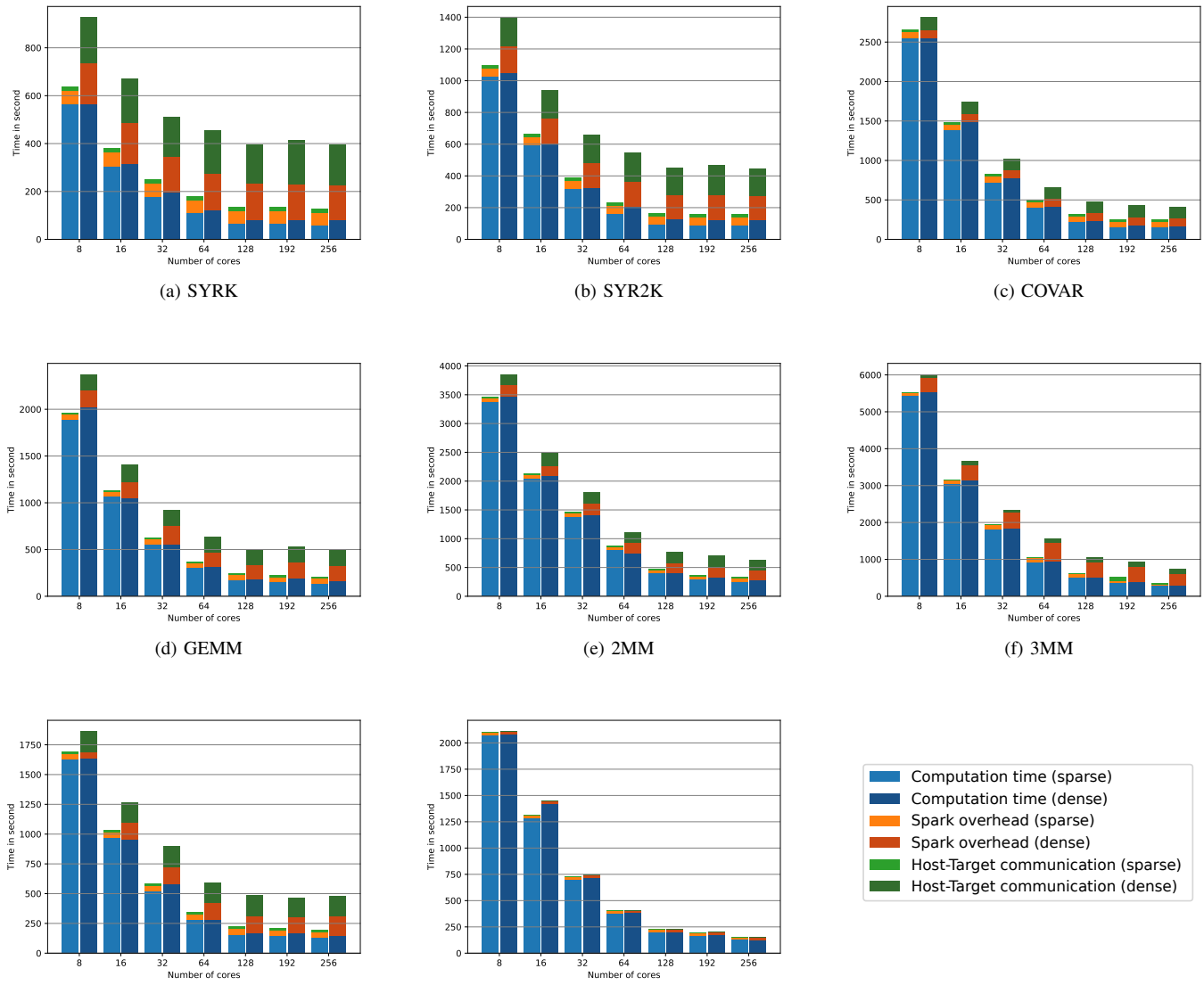
V. Related works

Cloud offloading has been largely studied for mobile computing in order to increase the computational capabilities of cellphones [29]. Some frameworks have been proposed to facilitate the development of mobile applications using cloud resources [30], [31], [32]. Contrary to our approach which relies on C/C++, they mostly rely on .NET or Java which are the most popular environments for mobile device. One of the key feature of those offloading framework, not treated in this paper, is to dynamically determine if it worth offloading the computation in term of communication overhead and energy consumption [33]. Some of those frameworks even explore the parallelization of the execution by providing multi-threading support, or virtual machine duplication, but it requires a considerable programming effort for non programming expert and their result do not present very large speedups (up to 4x). Additionally, older works used a similar offloading execution model to accelerate spreadsheet processing in grid computing [34], [35].

Recent works have been proposed to port scientific applications from various domains to private and public clouds [36], [37], [38], [39] by using a mixture of MPI and OpenMP; an approach that benefits from the communication efficiency of MPI primitives and the easy parallelization of OpenMP annotations. Experimental results usually show good performance, but they also reveal the difficulties associated to MPI programming, which require a level of expertise and platform knowledge that is far beyond the knowledge of typical programmers. Such drawbacks keep most programmers away from parallelizing their applications, constraining the computational power of the cloud cluster to a small set of expert programmers and specialized applications [11].

Several other works have proposed to use directive-based programming for programming computer clusters. First, Nakao et al. proposed a new directive-based programming language similar to OpenMP but specialized to HPC clusters [40]. Their directives allow to micromanage parallelization and communication within the distributed architecture of the cluster; their custom compiler then translates clauses into MPI calls. The OpenACC accelerator model is used to offload computation to the GPU within each node. If on one hand, their work demonstrates very good scalability, close to hand-made MPI implementation, on the other hand, their extensions do not follow the OpenMP standard, and require information about the cluster architecture (e.g. the number of nodes) which reduces its portability. Second, Jacob et al. introduced a new methodology relying on the OpenMP accelerator model to run applications on a cluster using the MPI infrastructure [22]: the host code is executed by the master node and the offloaded kernels by the worker nodes. Offloading to a set of worker nodes is then achieved by defining the offloaded kernel inside a *parallel loop* body. However, this requires various hand-made modifications to applications, such as programmatically splitting (and merging) the offloaded data and defining nested DOALL loop to parallelize the execution on multicore worker nodes. Finally, Wottrich et al. [41] proposed a set of new OpenMP directives based on a Hadoop MapReduce framework to extend the standard towards cloud computing. Similarly to our work, they consider the offloading from local computer to the cloud. Although their approach was supported by a set of proofs-of-concept, their code transformation was performed by hand and did not include any evaluation of the communication overhead. Moreover, they defined a new syntax for mapping variables to the cloud which is not compliant to the OpenMP and C standards.

Unlike previous works, our approach aims at providing the cloud as an additional accelerator available directly from the computer of the programmer while respecting the OpenMP standard. Next, we fully implemented our approach in existing tools allowing us to experiment it on a set of benchmarks and



(a) SYRK (b) SYR2K (c) COVAR

(d) GEMM (e) 2MM (f) 3MM

(g) Mat-mul (h) Collinear-list

Fig. 5: Average load distribution of cloud offloading according to the total number of worker cores and the data type.

to analyze the performance cost involved in cloud offloading. Last, contrary to most of previous works which rely on MPI, we rely on Spark a modern framework that has already been extensively used in the industry and is supported by a very dynamic community.

VI. Conclusion

In this paper, we addressed the problem of offloading computation to cloud infrastructures so as to benefit from their quasi-unlimited parallel processing capabilities. We choose to base our methodology on a directive-based programming paradigm because of its simplicity. In order to ease the utilization of the cloud, we designed a runtime that offloads, maps and schedules computation automatically. Our approach allows portability over commercial cloud services and private clouds. Indeed, by using a configuration file, our runtime is able to easily switch from one infrastructure to another without recompiling the binary (assuming compatible instruction-sets). The communication with cloud storage service and the execution within the Spark cluster is handled automatically according to the given configuration. Experiments were performed on a set of benchmarks described using the OpenMP accelerator model. Our results show that, because of data compression, the overhead induced by the offloading and the distributed computation heavily depends on the type of data processed by the application. Finally, promising performance was demonstrated by good speedups, reaching up to 86x on 256 cloud cores for the *2MM* benchmark using 1GB matrices. In the future, we plan to implement data caching to limit the cost of host-target communications.

ACKNOWLEDGMENTS

This work is supported by CCES CEPID/FAPESP under process 2014/25694-8. The experiments are also supported by the AWS Cloud Credits for Research program.

REFERENCES

[1] L. Lamport, "The parallel execution of do loops," *Commun. ACM*, vol. 17, no. 2, pp. 83–93, Feb. 1974.

[2] R. Cytron, "Doacross: Beyond vectorization for multiprocessors," in *ICPP*, 1986.

[3] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, "Automatic thread extraction with decoupled software pipelining," in *MICRO '38*, 2005.

[4] J. R. Allen, "Dependence analysis for subscripted variables and its application to program transformations," Ph.D. dissertation, Rice University, 1983.

[5] M. S. Lam and M. E. Wolf, "A data locality optimizing algorithm," *SIGPLAN Not.*, vol. 39, no. 4, pp. 442–459, Apr. 2004.

[6] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, Oct 1991.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplied Data Processing on Large Clusters," in *Proceedings of 6th Symposium on Operating Systems Design and Implementation*, 2004, pp. 137–149.

[8] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the mpi message passing interface standard," *Parallel computing*, vol. 22, no. 6, pp. 789–828, 1996.

[9] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[10] OpenMP, "OpenMP Application Program Interface," Tech. Rep., 2013.

[11] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A Berkeley view of cloud computing," University of California, Berkeley, Tech. Rep., 2009.

[12] I. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. Ullah Khan, "The rise of "big data" on cloud computing: Review and open research issues," *Information Systems*, vol. 47, pp. 98–115, 2015.

[13] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, 2010, pp. 1–10.

[14] C. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," *Advances in neural information processing systems*, vol. 19, p. 281, 2007.

[15] A. McKenna, M. Hanna, E. Banks, A. Sivachenko, K. Cibulskis, A. Kernytsky, K. Garimella, D. Altshuler, S. Gabriel, M. Daly *et al.*, "The genome analysis toolkit: a mapreduce framework for analyzing next-generation dna sequencing data," *Genome research*, vol. 20, no. 9, pp. 1297–1303, 2010.

[16] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark : Cluster Computing with Working Sets," in *HotCloud'10 Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, p. 10.

[17] T. Hunter, T. Moldovan, M. Zaharia, S. Merzgui, J. Ma, M. J. Franklin, P. Abbeel, and A. M. Bayen, "Scaling the mobile millennium system in the cloud," *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11*, pp. 1–8, 2011.

[18] H. Zhang, J. Yan, and Y. Kou, *Efficient Online Surveillance Video Processing Based on Spark Framework*. Cham: Springer International Publishing, 2016, pp. 309–318.

[19] D. Teijeiro, X. C. Pardo, P. González, J. R. Banga, and R. Doallo, *Implementing Parallel Differential Evolution on Spark*. Springer International Publishing, 2016, pp. 75–90.

[20] M. S. Wiewiorka, A. Messina, A. Pacholewska, S. Maffioletti, P. Gawrysiak, and M. J. Okoniewski, "SparkSeq: Fast, scalable and cloud-ready tool for the interactive genomic data analysis with nucleotide precision," *Bioinformatics*, vol. 30, no. 18, pp. 2652–2653, 2014.

[21] OpenACC, "The OpenACC Application Programming Interface," Tech. Rep., 2013.

[22] A. C. Jacob, R. Nair, A. E. Eichenberger, S. F. Antao, C. Bertolli, T. Chen, Z. Sura, K. O'Brien, and M. Wong, "Exploiting fine- and coarse-grained parallelism using a directive based approach," in *Lecture Notes in Computer Science*, vol. 9342, 2015, pp. 30–41.

[23] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International Symposium on Code Generation and Optimization (CGO)*, no. c, 2004, pp. 75–86.

[24] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, 1996.

[25] J. P. Hoeflinger, "Extending OpenMP to Clusters," *Intel Corporation white paper*, 2006.

[26] Y. C. Hu, H. H. Lu, A. L. Cox, and W. .Zwaenepoel, "{OpenMP} for networks of {SMP}s," *Journal of Parallel and Distributed Computing*, vol. 60, no. 12, pp. 1512–1530, 2000.

[27] L.-N. Pouchet, "Polybench: The polyhedral benchmark suite," http://web.cse.ohio-state.edu/~pouchet/software/polybench/, May 2015.

[28] F. Magno Quintao Pereira, D. do Couto Teixeira, K. Andrade, and G. Souza, "Mgbench: Openacc benchmark suite," https://github.com/lashgar/ipmacc/tree/master/test-case/mgBench.

[29] K. Kumar, J. Liu, Y. H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile Networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[30] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, 2010, pp. 49–62.

[31] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," *Proceedings - IEEE INFOCOM*, pp. 945–953, 2012.

[32] M. Gordon, D. Jamshidi, and S. Mahlke, "COMET: code offload by migrating execution transparently," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012, pp. 93–106.

[33] M. V. Barbera, S. Kosta, A. Mei, and J. Stefa, "To offload or not to offload? the bandwidth and energy costs of mobile cloud computing," in *Proceedings - IEEE INFOCOM*, 2013, pp. 1285–1293.

[34] K. Nadiminti, Y. Chiu, N. Teoh, A. Luther, S. Venugopal, and R. Buyya, "ExcelGrid: A .NET plug-in for outsourcing Excel spreadsheet workload to enterprise and global grids," in *Proceedings of the 12th International Conference on Advanced Computing and Communication*, 2004.

[35] D. Abramson, J. Dongarra, E. Meek, P. Roe, and Z. Shi, "Simplified grid computing through spreadsheets and NetSolve," in *Proceedings - Seventh International Conference on High Performance Computing and Grid in Asia Pacific Region, HPCAsia 2004*, no. August, 2004, pp. 19–24.

[36] M. Nikolić, M. Hajduković, D. D. Milašinović, D. Goleš, P. Marić, and Ž. Živanov, "Hybrid MPI/OpenMP cloud parallelization of harmonic coupled finite strip method applied on reinforced concrete prismatic shell structure," *Advances in Engineering Software*, vol. 84, pp. 55–67, 2015.

[37] M. Hajduković, D. D. Milašinović, D. Goleš, M. Nikolić, P. Marić, Ž. Živanov, and P. S. Rakić, "Cloud Computing based MPI/OpenMP Parallelization of the Harmonic Coupled Finite Strip Method applied to Large Displacement Stability Analysis of Prismatic Shell Structures," *Proceedings of the Third International Conference on Parallel, Distributed, Grid and Cloud Computing for Engineering*, pp. 1–21, 2013.

[38] V. Nikl and J. Jaros, "Parallelisation of the 3D Fast Fourier Transform Using the Hybrid OpenMP/MPI Decomposition," in *Mathematical and Engineering Methods in Computer Science*, 2014, vol. 8934, pp. 100–112.

[39] R. D. Haynes and B. W. Ong, "MPI-OpenMP Algorithms for the Parallel Space-Time Solution of Time Dependent PDEs," in *Domain Decomposition Methods in Science and Engineering XXI*, 2014, pp. 179–187.

[40] M. Nakao, H. Murai, T. Shimosaka, A. Tabuchi, T. Hanawa, Y. Kodama, T. Boku, and M. Sato, "XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters," *Proceedings of WACCPD 2014: 1st Workshop on Accelerator Programming Using Directives*, pp. 27–36, 2015.

[41] R. Wottrich, R. Azevedo, and G. Araujo, "Cloud-based OpenMP Parallelization Using a MapReduce Runtime," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2014 IEEE 26th International Symposium on*, 2014, pp. 334–341.