GRU: Exploring Computation and Data Redundancy via Partial GPU Computing Result Reuse

Husheng Zhou
The University of Texas at Dallas
husheng.zhou@utdallas.edu

Soroush Bateni
The University of Texas at Dallas
soroush@utdallas.edu

Cong Liu
The University of Texas at Dallas
cong@utdallas.edu

ABSTRACT

Graphics processing units (GPUs) have been widely adopted by major cloud vendors for better performance and energy efficiency. Recent research has observed a considerable degree of redundancy in managing computation and data in many datacenters, particularly for several important categories of GPU-accelerated applications such as log mining and machine learning. In this paper, we present GRU, an ecosystem that smartly manages and shares GPU resources through exploiting redundancy. GRU transparently interprets GPU-accelerated computing requests and memoizes results for potential future reuse. To enhance reusability, GRU implements a partial result reuse idea, where GPU computation requests even with different input data and functionality may become reusable w.r.t. each other. To guarantee correctness of partial reuse, GRU employs a compiler-assisted approach that analyzes general data parallel patterns that are reliable for the reuse purpose, and is capable of smartly recognizing such reusable data parallel patterns of incoming requests. We have fully implemented GRU and conducted extensive sets of experiments running micro-benchmarks on local machines and real-world applications including Spark-based uses cases in an AWS cluster. Evaluation results show that GRU is effective in identifying and eliminating redundant GPU computations, achieving up to 5x (2.5x) speedup for compute-intensive (data-intensive) benchmarks. In addition, GPU-managed Spark observes a reduction of 25.3% (39.8%) on average w.r.t. turnaround time (GPU occupation time) over state-of-the-art solutions.

CCS CONCEPTS

• **Computing methodologies** → **Parallel computing methodologies**;

KEYWORDS

GPU, Partial Result Reuse, Redundancy

ACM Reference Format:

Husheng Zhou, Soroush Bateni, and Cong Liu. 2018. GRU: Exploring Computation and Data Redundancy via Partial GPU Computing Result Reuse. In *ICS '18: International Conference on Supercomputing, June 12–15, 2018, Beijing, China*. ACM, New York, NY, USA, Article 4, 10 pages. <https://doi.org/10.1145/3205289.3205318>

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ICS '18, June 12–15, 2018, Beijing, China

© 2018 Association for Computing Machinery.

ACM ISBN 978-1-4503-5783-8/18/06...\$15.00

<https://doi.org/10.1145/3205289.3205318>

1 INTRODUCTION

Graphics processing units (GPUs) are powerful accelerators for compute-intensive data-parallel applications. With existing GPU programming models and runtime environments such as NVIDIA's *compute unified device architecture* (CUDA) [34], GPUs have been widely adopted by datacenters and cloud vendors [1, 15, 21, 32] for accelerating general-purpose workloads (*a.k.a.*, GPGPU) with better performance and energy efficiency.

A large degree of computing redundancy has been observed in datacenters and cloud, particularly in a multi-tenant environment sharing similar computing interests on the same or incremental datasets [4, 5, 36], as also highlighted by Nectar [17]. With the trend of widely adopting GPUs in such environments, we seek to exploit computing and data redundancy specifically on GPU through reusing GPU computing results for any redundant requests. Compared to CPU-based computation reuse, reusing results processed on GPUs may be more viable due to the nature of GPU processing. Employing GPUs for general-purpose computing (e.g., using CUDA) is usually a four-step procedure: transferring kernel code binary to GPU, copying input data to GPU, launching the kernel, and acquiring the output. This four-step procedure makes GPGPU a “blackbox”—with the same input data (due to data redundancy) and the same computing kernel (due to functional redundancy), GPU will always generate the same results in most cases (except for explicit random number generation function calls).

The following facts further motivate us to explore the appealing GPU computing result reuse idea: (i) low-level interfaces for accessing GPU are unified on current commercial products (e.g., CUDA), (ii) GPU computing codes and input data are explicitly extracted and transferred to GPU, making them easier to intercept and identify, and (iii) most scientific computing researchers prefer to use standard, highly-optimized libraries provided by GPU vendors (e.g., cuBLAS library provided by NVIDIA for matrix operations). These imply that a good standardization of GPGPU interfaces may greatly increase the degree of redundancy seen in GPGPU computing requests (also see our measurements-based case studies using Spark in Sec. 2). Since workloads offloaded to GPU are often computation-intensive, eliminating redundant computations through result reuse may lead to significantly better performance in terms of latency, resource utilization, and energy efficiency.

Unfortunately, it is not straightforward to directly apply this idea as any exact result reuse requires the computing code and input data of two computations to be exactly the same. This requirement may not be easily met in practical settings since even a tiny bit of mismatch in input data or computing code would cause two GPU computations to be different. Thus, in this paper, we focus on resolving this challenging issue of smartly and correctly reusing GPU computing results.

Contributions. This paper presents GRU— a GPU sharing, result memoization and reuse ecosystem. A key factor that determines the efficiency and practicality of our GPU computing result reuse idea is clearly reusability, i.e., the amount of computations that may be reusable among all computing requests. Instead of employing the straightforward approach of exact result reuse which requires any result to be reusable only if two requests have exactly the same computing code and input data, GRU develops a core technique of partial result reuse. This significantly enhances reusability as kernel launches even with different input data may become reusable w.r.t. each other as long as their input data share common portions. Specifically, GRU employs a compiler-assisted approach to smartly partition input data of a computing request into smaller sub-chunks such that any result produced corresponding to a data sub-chunk becomes reusable. A challenge herein is to determine whether an input data can be partitioned correctly and how to perform the partition efficiently. To resolve this, we analyze general data parallel patterns that are reliable for the reuse purpose, and design GRU in a way such that it is capable of smartly recognizing such reusable data parallel patterns of incoming requests.

GRU is composed of a frontend (FE) deployed at user-side which consists of a rewriter and a library, and a backend (BE) deployed as a cloud/cluster wide service that receives GPU computation requests from FE and caches derived computation results. A core component is a reuse engine integrated into the BE, which performs a check on each incoming GPU computing request to decide whether or not the result of any previously processed request is reusable. A result cache is used to store previously computed results, which is a fast key-value store that indexes each resultant data with a fingerprint of its corresponding input (i.e., using hash values of input data chunks) and the fingerprint of the computing kernel (i.e., the hash value of kernel code binary).

Our design of GRU is comprehensive in the sense that GRU supports a wide range of GPGPU applications ranging from single kernel applications to complicated ones composed of kernel graphs with data dependencies. We have implemented GRU in a multi-node AWS cluster and a desktop computer, and conducted an extensive set of experiments using micro benchmarks from different domains, and rather complicated programs of a big data processing framework, Spark [22, 49]. The evaluation demonstrates that (i) GRU can achieve a better performance, up to 5x speedup for compute-intensive and up to 2.5x speedup for data-intensive benchmarks, (ii) Our design and implementation of GRU is reliable enough to support rather complicated large-scale programs as it supports the Apache Spark with a considerable performance improvement both in terms of turnaround time (25.3% on average) and GPU occupation time (39.8% on average), and (iii) Our current implementation of GRU is reasonably efficient, with less than 7% run-time overhead incurred for most of the evaluated benchmarks.

Application scope of GRU. We design GRU to be deployed in a dedicated cloud/cluster, where multiple tenants are interested in processing workloads from similar application domains (e.g., making earthquake/weather prediction in an HPC cluster). Several important categories of GPU-accelerated applications may greatly benefit from GRU, such as machine learning (e.g., nearest-neighbor, etc.) and log mining (e.g., words statistics, etc.). In general, environments that may significantly benefit from applying GPU are

Table 1: Movie recommendation using GPU-enabled Spark on two movieLens datasets with different partitioning.

Datasets	Block	Dim (K)	Launch	Redundancy	
movieLens-20M	1	1650x288	1	0	0%
movieLens-26M	1	5290x920	1		0%
movieLens-20M	144	11.5x2	156	88	56.41%
movieLens-26M	460	11.5x2	483		18.22%
movieLens-20M	2304	2.8x0.5	2352	2112	89.80%
movieLens-26M	7520	2.8x0.5	7614		27.74%

those in which users run the same set of deterministic functionalities (i.e., not incurring random instructions such as PRNG kernels) on similar or incremental datasets (e.g., researchers performing various text analytical operations on trace logs in a data center).

2 CASE STUDY

We conduct a measurement-based case study to motivate the potential benefit of result reuse in GPGPU computing. An error-free reuse would leave no room for approximate memoization techniques. This requirement combined with the blackbox nature of GPGPU implies that the entire input data and the entire kernel code should be an exact match for any possibility of reuse. Throughout this paper, we use a measurement called redundancy to deduce the final possibility of reuse for GPU computations. Redundancy is defined as two equivalent GPU kernel launches with the exact same input data and kernel code.

We use a publicly available dataset movieLens [18], which is a collection of rating data from the MovieLens web site. We used two versions of movieLens: one denoted “movieLens-20M” released in 2015 which contains 20 million ratings applied to 27,000 movies by 138,000 users; the other one denoted “movieLens-26M” released in 2017 which contains 26 million ratings applied to 45,000 movies by 270,000 users [18]. We run a commonly performed matrix factorization function [23] on both versions of movieLens. For these datasets, the corresponding matrix is a $27,000 \times 138,000$ and a $45,000 \times 270,000$ sparse matrix, respectively. The first two rows of Table 1 show the results for an unabridged redundancy among these two datasets. Since these two databases are different, the redundancy is 0, implying that there does not exist any pair of GPU computing requests exhibiting exactly the same input data.

Next, we set out to test the effectiveness of a straightforward implementation of partial result reuse that divides the input data into smaller chunks. We use a popular big data processing infrastructure—Spark [22] to process movie recommendations. The Spark framework is in charge of running a map-reduce algorithm, in turn partitioning the matrix into smaller blocks to improve parallelism on different workers.

For each dataset, we performed two runs with different partitioning parameters, depicted in the rest of Table 1. The first column of Table 1 distinguishes the datasets. The second column indicates the number of blocks the original dataset is partitioned into, and the dimension of each block (in Kilo rows/columns per block) is shown in the third column. The fourth column is the number of invoked GPU kernel launches. The last column indicates the redundant GPU computation between the two runs with the same dimension. For example, when we partition datasets into $11.5K \times 2K$ blocks, the

Figure 1: GRU architecture consists of a rewriter and a library at the front-end, and a back-end service that runs in the cluster.

movieLens-20M dataset is partitioned into 144 blocks and invokes 156 GPU kernel launches, while the movieLens-26M dataset is partitioned into 460 blocks invoking 483 kernel launches. Among these two runs, 88 kernel launches are redundant, meaning that the input data and computation are equivalent, accounting for 56.41% and 18.22% of total kernel launches within each run respectively.

Through data partitioning, we observe a good degree of redundancy between the two runs. This is intuitive as movieLens-26M is similar in nature to movieLens-20M, implying that they contain a large fraction of equivalent data blocks. Furthermore, when we further partition these datasets into finer-grained $2.8K \times 0.5K$ blocks, the redundant computations account for 89.8% and 27.74% among the two runs. This is because with a finer-grained block size, more fractions of the datasets can potentially become equivalent.

This case study reveals a considerable redundancy in production runs on similar or incremental datasets. Such redundancy is expected to be even more significant in large-scale functionality-dedicated clusters/datacenters where thousands of users/applications perform computation requests of common interests [17]. This case study also highlights the following critical observation: rather than relying on exact result reuse which may suffer from low redundancy due to coarse-grained consideration of input data, exploring partial result reuse on GPU could potentially elevate the reuse idea to become effective in practical settings due to increased redundancy and reusability.

3 GRU DESIGN

3.1 Overview

Our core idea behind GRU is to avoid redundant computations on GPU. This is achieved by reusing the results of previous computations that exhibit the same computing code and input data. The architecture of GRU is illustrated in Fig. 1. GRU consists of a front-end that runs on the user’s desktop or an instance in the cloud, and a back-end service running on a cluster or another instance(s) in the cloud. The front-end is composed of a rewriter and a library which are shown as shadowed rectangles. The back-end consists of a reuse engine and a result cache.

GRU is designed to be transparent to end-users. End-users shall transparently use the GRU rewriter to compile a CUDA source code and run the generated binary executable as a normal CUDA program. At runtime, the GRU library forwards the GPU computing requests including GPU memory allocations and kernel launches

Figure 2: Two similar images (a) (b) with same tile (c).

to the reuse engine located in the back-end. The results are then retrieved by GRU and returned to the program. The CUDA program can be run on a physical desktop or a virtual instance without a native GPU hardware since all the actual GPU computations are launched on the back-end.

The reuse engine that resides in the GRU back-end is a key component that is in charge of processing incoming GPU computation requests and relaying the output data back to the front-end. The reuse engine uses the hash value of the GPU binary (i.e., .cubin) as the fingerprint of a GPU kernel (i.e., a piece of GPGPU computing code). In addition, the reuse engine uses the hash value of the data chunk as the input’s fingerprint. Upon receiving a kernel launch, the reuse engine calculates hash values (of kernel’s fingerprint, input’s fingerprint, and other primitive type parameters) and consults the result cache. The result cache implemented using a LUT stores previous kernel launch results in the form of metadata. Once the fingerprint is found in the LUT, the actual GPU computations are skipped and the cached result is directly reused. Otherwise, the reuse engine issues the kernel launch request to GPU and relays the output to the front-end. In the meantime, this output is inserted into the result cache as a new entry.

In our current implementation and evaluation, the reuse engine and the result cache components are located on the same physical machine which is equipped with physical GPUs. However, the result cache can be easily decoupled from the reuse engine and put onto a dedicated cache server, providing cloud/cluster-wide result caching over the network.

3.2 Methodology

3.2.1 Reuse Basics. Many GPGPU programs contain a single kernel that is transferred to GPU from main memory. The kernel is then launched on the GPU accompanied by its parameters. GRU treats the result of a single kernel launch as the basic reusable unit. To realize result reuse, GRU needs to first interpret and identify each incoming kernel’s code and parameters. If a parameter is of GPU memory pointer type, GRU would need to dereference the pointer in order to obtain the content. The kernel code and the input data can be identified by GRU through intercepting CUDA runtime APIs.

After obtaining the kernel code and input data, GRU dereferences the pointer parameters and obtains the hash value (fingerprint) as the key to probe result cache. If there is a corresponding cached result (cache-hit), it is used as the output of the kernel launch. In the case of a cache-miss, the kernel launch request is forwarded to the CUDA driver for the actual computation on GPU. The computed result along with the fingerprint are then stored in result cache as a new entry.

Figure 3: Three example data parallel patterns: (a) Map (b) Partition (c) Scatter/Gather.

3.2.2 Partial Result Reuse via Smart Data Partitioning. Under the above-mentioned basic reuse principles, the result of a kernel launch can be reused only if an entry can be found in the result cache that contains the exact same kernel code and parameters. However, for many scenarios in practice, multiple users may execute the same GPGPU kernel program using similar but not exactly the same input data. Fig. 2 shows an example scenario, where two similar images (shown in Fig. 2(a) and (b)) are being analyzed using the same image convolution kernel. By applying the exact reuse principle, these two launches are not reusable w.r.t. each other. However, an interesting observation is that there exists a common tile between these two images, as is shown in Fig. 2(c). Thus, the result of the same kernel launch using this tile as the input data becomes reusable.

Similar to content-based data partitioning applied in deduplication systems in the context of storage and incremental computation frameworks [4, 5], our intuitive idea is to allow GRU to selectively pre-partition a request’s input data into smaller chunks, thus increasing the redundancy and reusability through partial result reuse. GRU is designed to realize the idea of partial reuse by analyzing the kernel code provided by the user and transparently performing a compiler-assisted transformation. The transformation is responsible for converting a normal GPU program into one that can communicate with GRU by injecting GRU API calls wherever deemed necessary. A challenge herein is to determine whether an input data can be correctly partitioned and how to efficiently perform the partition. To resolve this challenge, GRU analyzes general data parallel patterns that are reliable for the reuse purpose, and is capable of recognizing such reusable data parallel patterns of incoming requests at runtime.

According to McCool and Samadi [30, 40], data parallel patterns can be generally categorized into six types: map, partition, scatter/gather, reduction, scan, and stencil (see [30, 40] for detailed definitions of these patterns). A key observation is that applications containing the map and partition patterns (as illustrated in Fig. 3(a) and (b)) are well suited for input data partitioning and consequently, for partial result reuse purposes. Intuitively, this is because under these two patterns, the elements of the output can be separated into groups that are independent of each other. As for application exhibiting other data parallel patterns, it is not suitable to perform data partitioning due to rather complicated dependencies. For example, for the scatter/gather pattern as illustrated in Fig. 2(c), every element in the input array is randomly accessed by the kernel to produce an output element, making data partitioning unsuitable.

This limitation does not impair GRU’s ability to be effective in most real scenarios since a large portion of parallel tasks, applications that use map/reduce processing, exhibit the patterns of map and partition [13, 20, 46]. However, the remaining four patterns are not entirely dismissed by GRU. Even if an application contains

```

1  TYPE* a = (TYPE*)malloc(sizeof(TYPE));
2  TYPE* b = (TYPE*)malloc(sizeof(TYPE));
3  TYPE* c = (TYPE*)malloc(sizeof(TYPE));
4  init_data(); // init value of a, b, c
5  struct matrix_handle_t A_handle, B_handle, C_handle;
6  matrix_partition (&A_handle, a, sizeof(TYPE),
7                  slice_a, // num of slices
8                  xdim_a, ydim_a, // dimension
9                  vertical_filter); // filter function
10 matrix_partition (&B_handle, b, sizeof(TYPE), slice_b,
11                  xdim_b, ydim_b, horizontal_filter);
12 matrix_map_filter (&C_handle, c, sizeof(TYPE),
13                  vertical_filter, slice_a,
14                  horizontal_filter, slice_b);
15 gru_launch (&kernel, A_handle, B_handle, C_handle, ...)

```

Figure 4: Code segment of matrix multiplication program after transformation.

scatter/gather, reduction, scan, and stencil, there is still a chance for the output to be reused if the output is transcribed using a map or partition pattern. For example, the kernel of a matrix multiplication combines a row and a column of elements of the input matrices into a single element for the output matrix. This operation exhibits the reduction pattern that has inherent dependencies. However, the output is written as independent non-overlapping sections spanning one node which is the partition pattern [30]. Thus, the input data of a matrix multiplication kernel can be safely partitioned using a row or a column as the smallest partition unit since this scheme of data partitioning will not interfere with the reduction. In other words, it is safe to break up the computations (through data partitioning) that exhibit the partition pattern because these computations are independent. However, breaking up the computations of a reduction pattern will result in an invalid state.

GRU introduces a compiler-assisted approach to identify kernels that allow for data partitioning and perform the proper partitioning at compile time. We setup an LLVM [26, 47] pass to analyze the IR (intermediate representation) of both CUDA kernel code and CPU code. This will determine the access patterns of the input and output data in the kernel. GRU associates the memory access of each memory object with the thread’s built-in parameters, such as blockIdx and threadIdx whose value ranges can be inferred according to the dimensions of each kernel. Then these parameters and their ranges are converted into STP-recognizable constraints and sent to the STP solver [43]. The STP solver decides whether or not one individual element (map pattern) or a fixed set of elements (partition pattern) are accessed by every thread exclusively. More precisely, GRU can find out if the fixed set of elements are in vertical, horizontal, or block manner. Thus, we can get the partition plan accordingly. Specifically, a kernel is determined to be feasible for partitioning if it contains the aforementioned data access patterns. The compiler pass will then insert a set of GRU APIs that perform the data partitioning using a set of predefined partition filters: `vertical_filter`, `horizontal_filter`, and `block_filter` for up to three dimensional data arrays. Fig. 4 illustrates an example, showing the code segment of the matrix multiplication kernel after transformation (before the transformation, the code only consists of a kernel launch). We note that this code segment is only used for demonstration purposes. In reality, the GRU rewriter performs the code transformation using the LLVM IR and sends the IR to

Figure 5: Different functionalities share common sub-computation K1 and K2 that can be reused from previous cached results.

the LLVM backend to generate the machine code directly. As is evident in the figure, Lines 5 - line 11 convert the original input data chunks into partitionable data structures. Lines 12 - 14 partition the output data chunks according to the filters applied to the input matrices. Actual data movement and kernel launch happens when `gru_launch` is called at line 15. The detailed analysis and rewriting algorithm will be discussed next in Sec. 4.

3.2.3 Partial Result Reuse via Sub-Functionality. While we have covered partial reuse given the same computation code so far, it is actually possible to partially reuse results from different input data and different computation codes. The constraint is the fact that some sub-component of the computational flow is shared across two executions, along with a partially reusable input data.

GRU is designed to consider GPU kernel dependency graphs and reuse parts of the execution flow accordingly. Fig. 5 illustrates such a scenario where the entire computation serves a single purpose and is depicted in the black square. However, on the inside, the computation consists of multiple sub-kernels for Fig. 5(a), the kernels are K1, K2, K3, and K4. For Fig. 5(b), the kernels are K1, K2, K5, K6, and K7. Fig. 5(b) depicts the scenario in which the computing results of K1 and K2 are previously cached by GRU. Thus, their computations can be skipped by directly using the cached results and copying them to GPU device memory to be used as K7's inputs.

3.3 GRU Front-End

3.3.1 Rewriter. As was mentioned above, one major role of the rewriter is to transform the original program into a partition-enabled one. Beyond this capability, the rewriter is also in charge of: 1) Identifying the inputs and outputs of each GPU kernel; and 2) Deciding whether to rewrite the vanilla GPU kernel launch as a reuse-enabled kernel launch. We design the Rewriter to be transparent to the user. We employ an intermediate compile-time technique detailed in Sec. 4. The major role of Rewriter is to intervene at compile time and redirect API calls to the back end through the front-end library. We describe this library and the back-end next.

3.3.2 Front-end Library. GRU front-end library resides in the user-side which is a substitute for the CUDA runtime and computation libraries (e.g., `libcublas.so`). Front-end library intercepts all CUDA calls made by applications, and forwards each request to the BE. It is also in charge of managing the connection between the user's desktop/instance and BE. After being loaded by CUDA applications, the front-end library automatically establishes a connection with BE that is specified through an environment variable.

For each CUDA call, FE sends the request together with the related parameters to BE, and awaits the response.

Moreover, high performance computing applications often require massive data transfers between the CPU main memory and GPU device memory. In GRU, since all interactions between CPU main memory and GPU device memory occur in backend, such massive data does not need to be transferred between FE and BE. The GPU device memory pointers are never de-referenced on the user-side. Rather, the actual GPU device memory pointer is transferred between the guest and the host in the form of a hex value. Only if a guest memory pointer takes part in a computation on the host the memory chunk will be transferred over.

3.4 GRU Back-End

The GRU Back-end is located in the cluster/cloud, and is responsible for executing CUDA calls received from the FE and returning the computed results. For each application in FE, a new process is created on the host to run all the CUDA requests related to that application using an independent GPU context. BE then forwards the requests received from the FE to the CUDA device driver running on the host physical machine. Finally, BE transfers the computed outputs back to FE at user-side.

The Reuse Engine is implemented in the BE as a core component to realize our idea of result reuse, and acts as a bridge between BE and the CUDA runtime libraries. The reuse engine selectively caches GPU computation results and maintains them in a result cache. We use the Sqlite3 in-memory database to store the cache results. Sqlite3 is open-source and widely used in practice. Moreover, Sqlite3's reliability and performance can be guaranteed [19]. We choose an in-memory database mainly due to performance concerns, since the LUT would be retrieved and updated frequently, at least once for every kernel launch request.

The latency of hashing operations is rather small, even with increased data sizes. We use an extremely fast non-cryptographic hash algorithm to perform the hashing [48], the speed of which can reach 13.8 GB/s on a 64 bits OS. The possibility of collision is also guaranteed by the hash algorithm ($1 \sim 1.8 \times 10^{19}$). The latency of the search and update operations is less than 0.05 ms which is trivial compared to a typical kernel execution time (typically > 100 ms). The dump operation is occurred only when a cache-hit occurs. Thus, compared to the actual kernel execution time, time-saving is still achieved. On the other hand, the result insertion operation is rather time-consuming, which also in turn increases with the result size. However, such operation does not affect the user application's response time since it runs asynchronously with the computing results transfer from reuse engine to FE.

By intercepting the incoming requests from FE, the reuse engine decides whether to issue requests to the CUDA runtime or directly reuse cached results. In our current implementation and evaluation environment, the reuse engine is located in the host machine which is equipped with physical GPUs. However, the fact is that the reuse engine can be easily deployed on a dedicated physical server in the cloud. The only requirement is that this machine should contain a sufficiently large RAM for the in-memory database. This machine would serve as a mem-cache server and would provide a cloud-wide result caching and reusing over the network. The detailed workflow of reuse engine is described next in Sec. 4.

Algorithm 1 Rewriting Algorithm

Require: gridDim ▷ Dimension of grid
Require: blockDim ▷ Dimension of thread block
Require: kernel ▷ GPU kernel to be analyzed

```

1: function PARTITIONPLAN(kernel, gridDim, blockDim)
2:   para[] ← list of pointer parameters of kernel
3:   plan[] ← return value
4:   for p in para[] do
5:     idx[] ← {} indexes of element access
6:     for ins in use_chain(p) do
7:       i ← associateBuildin(operand(ins), gridDim, blockDim)
8:       idx.push(i)
9:       g.push(pathConstrant(ins))
10:    if stpCheck(g, idx) is OVERLAP then
11:      plan[p] ← NONE
12:    else
13:      if typeOf(idx) is VERTICAL then
14:        plan[p] ← VERTICAL
15:      else if typeOf(idx) is HORIZONTAL then
16:        plan[p] ← HORIZONTAL
17:      else if typeOf(idx) is BLOCK then
18:        plan[p] ← BLOCK
19:      else
20:        plan[p] ← NONE
21:   return plan[]

```

4 IMPLEMENTATION DETAILS**4.1 Rewriting Algorithm**

Our implementation of the Rewriter is deeply embedded in the llvm compiler. The main task for Rewriter is to intercept IRs generated for both CPU and GPU codes. It will then run a separate pass to do a few analysis all at once. First, the Rewriter identifies any variable initialized in the CPU code. It will discard the ones that are not in any shape or form used in the GPU computation. For example, if a variable is allocated by cudaMalloc, then it is considered to be connected to GPU code. The Rewriter will also analyze the GPU code in order to determine the data pattern that exists in the GPU kernels and it can build the dependency graph of data objects.

The pseudo-code of this analysis is depicted in Alg. 1. The function PartitionPlan will iterate through the parameter list in lines 4-21. Lines 6-8 will get all the element access indexes for each parameter p . Line 9 will get the path condition required to reach the current instruction. Finally, lines 10-21 will decide the final pattern and directly choose a partition plan. For instance, the condition at line 10 will check for any possible overlap of parallel access pattern. If an overlap exists, Rewriter would conservatively discard the kernel as non-partitionable. The remainder of conditions in lines 14, 16 and 18 will check for our three data filters: vertical, horizontal and block filters. PartitionPlan will return a plan that will be used to take an informed action in later stages.

4.2 Result Cache and Reuse

The reuse engine is a key component in GRU BE, which receives kernel launch requests, checks reuse possibility, and returns the cached results or forwards requests to CUDA runtime for actual execution on GPUs. Fig. 6 depicts a detailed block diagram of the various components implemented in the reuse engine and shows the general operational flow. Specifically, a kernel launch request is first identified by the reuse engine through its input arguments and data chunk hash values. Then the launch request is filtered by the reuse engine according to its input size which is directly related to its computation time. We note that a GPU kernel may contain complex data structures as its parameters, which is hard to generate

Figure 6: Workflow of reuse engine

the fingerprint for. These parameters will not be considered for reusing purposes because such complex cases are rare [34].

The filtering process is important because reusing results for small kernels or kernels with a small input data size incurs a considerable overhead, which may be more time consuming than actually executing such kernels on GPU. For functions in libraries provided by the CUDA development kit, such as cublas and cufft libraries, we pre-define the reuse threshold through profiling. The guideline behind finding such threshold values is that reusing results shall yield less execution time than actually launching a kernel on GPUs. For example, we define the reuse threshold for cublasSgemm¹ to be 4MB, which implies that if the size summation of cublasSgemm's first two matrices is less than 4MB, we will not reuse the result.

If the launch request together with its inputs are worth reusing, then the reuse engine probes LUT to check if there are cached results; otherwise the launch request is directly sent to the CUDA runtime for actual computation. Each entry of the LUT contains the cached results and hash values of kernel code and input arguments. For any pointer argument, the hash value is the hash of its dereference. The hash value for a primitive type argument is itself. There is no semantic loss for arguments between guest and host, since the guest and host communicate through FE and BE which preserve the type information.

A hit occurs in the LUT when there is a cached result, in which case the reuse engine skips the actual computation on GPUs and directly dumps out the cached results to the result manager. When the result manager receives the GPU request for transferring back the result, it performs the actual result transfer. After an actual CUDA kernel launch, the results are supposed to be stored in the GPU device memory. There are three possibilities for such results depending on application scenarios: (i) results are transferred to host memory, (ii) results stay in the device memory for further computations, and (iii) results are copied to another location on the same GPU device memory. The result manager operates differently according to these three scenarios. Scenario (i) is the most common and straightforward. The result manager simply copies the cached results directly to the target host address. For scenarios (ii) and (iii), the cached results need to be transferred to the GPU device memory through additional CUDA calls for further computations, since the data on GPU is out-of-date (in (ii)) or invalid (in (iii)).

On the contrary, a miss in LUT indicates that result of the current computation request is not available for reuse. The computation

¹cublasSgemm is a CUDA linear algebra function in the CUBLAS library that perform matrix operations. It performs $A \times B + C$ where A, B and C are its three matrix inputs

request is then issued to the physical GPUs. A new entry is reserved for this computation in LUT. After the actual GPU computation completes, the corresponding entry will be updated accordingly.

If the reuse engine is deployed in a dedicated cache server, then cloud-wide GPU computing results can be pooled in such a server over the network, which increases the number of cached results and thus the reusability.

4.3 Global object tracking

We introduce the global object tracking optimization to track the content of GPU memory objects, in order to eliminate redundant data hashing of the same GPU memory object. For a majority of GPGPU computation requests, their parameters are in the form of GPU device memory pointers. In order to get the hash value of the kernel arguments, we need to first obtain the contents of the corresponding GPU memory objects. A clumsy method is to dump out the content of GPU objects, then perform the hash. However, under this method, for large objects, the GPU-to-CPU memory transfer may be quite time-consuming.

To eliminate such redundant hashing, GRU maintains a GOM (Global Object Map) to track the content of GPU objects, which is inspired by Moxie—a distributed dataflow engine for GPU object passing [39]. Each entry in GOM records a GPU object's handler, its corresponding CPU object, size, an up-to-date flag, shape, and hash value. An entry of a GPU object is registered to GOM when calling `cudaMalloc()`, and removed when calling `cudaFree()`. GRU uses GOM to keep the internal consistency of memory objects. If any CUDA request parameters are passed by the reference (typically a pointer), GRU can easily track the content of the GPU objects. Thus, when a GPU memory pointer is used for multiple kernel launches as the input parameter, it needs only to be hashed once. Besides, through GOM, we can easily identify a pointer which points to an arbitrary address within the memory space allocated to this object, according to the base address and size of a GPU memory object recorded in GOM. Redundant hashing can thus be further eliminated.

4.4 Delay transfer

The input data transferred through the network from the front-end to the back-end is unnecessary if a kernel launch can reuse the cached results. In a case like that, the input data is only used for hashing but is not needed to perform the actual computations. Consequently, for a large data requested by a `cudaMemcpyHostToDevice` call, we delay the actual transmission until the corresponding computation is launched, similar to the optimization in PTask [38].

If a computation request together with the hash value of the kernel code pass the reuse filter, the FE would first hash the input data chunk and transfers the hash value to BE instead of the entire data. If there is a cached result, then the actual input data does not need to be transferred. This delay operation is advantageous particularly for data-intensive applications.

5 EVALUATION

In this section, we first evaluate the performance gain when deploying GRU on Apache Spark and then evaluate GRU's performance breakdown and incurred runtime overhead using a set of popular micro-benchmarks.

5.1 Experimental Setup

In our evaluations, we use two different setups for Spark use cases and micro-benchmarks. For Spark experiment, we adopt GRU to SparkGPU [22] in a five-node cluster that consists of five AWS instances [1]. The cluster includes four CPU-only t2.xlarge instances and one GPU-powered g3.8xlarge instance that equip with two NVIDIA Tesla K80 GPUs. The GRU frontend residents on four CPU-only nodes and backend residents on the GPU node. When user launches a Spark program, four CPU nodes are used as workers, while the GPU computations are forwarded to the GPU node. For micro benchmarks, we adopt GRU on a desktop that equips with an i7-4790K 4.0 GHz processor, 128 GB memory and a NVIDIA Quadro 6000 GPU. To simulate the multi-tenants scenario, we use Xen hypervisor [3] to virtualize two instances (DomainU) as tenants. The GRU frontend residents in the instances, and the GRU backend is installed in Domain0. Benchmarks execute inside the instances, and the GPU computations are forwarded to and processed at the host machine (Domain0).

5.2 Spark Use Cases

We adopt GRU to SparkGPU [22] in a five-node cluster which includes four t2.xlarge instances and one g3.8xlarge instance. We conducted two sets of experiments. The first one investigates the benefit of applying GRU when different functionalities related to text mining are incurred upon the same dataset, while the second experiment studies the scenario where a commonly performed log analysis functionality is performed on different log traces.

5.2.1 Reuse with Different Functionalities. In this experiment we run four programs – WordStatistics, FreqWord, MostProduct, TopRatio on the public amazon review dataset [29]. WordStatistics records the number of each word's occurrences and the number of products that it appears in. FreqWord finds out the top K most frequently used words in the entire dataset. MostProduct looks for the top K words that appears in each category of products. TopRatio gets the ratio of top K mostly used words among all words. We use two versions of the dataset: a compact version with a size of nearly 8GB and a full version with a size of nearly 20GB. For each dataset, we enable GRU and run the WordStatistics program to cache the results. And the subsequent three programs can benefit from reusing cached results due to the WordStatistics program.

We measure the turnaround time (TAT) which indicates the time elapse from program launch till its completion, and GPU occupancy time (GOT) that is the accumulated GPU computation time on ever GPU device. Reducing GOT is crucial when GPUs are rare computing resources in a cluster, since GPUs can be yielded for other computing requests. As shown in Fig. 7, the x-axis represents three programs on two datasets. The y-axis is the execution time in minutes. The four histograms represents TAT without and with GRU on, and GOT without and with GRU on. The labels on top of the histograms indicate the percentage of the saved TAT and GOT. We observe that all three programs benefit from GRU even if they incur different functionalities, reaching an average of 25% TOT reduction and 39% GOT reduction. Furthermore, with a larger dataset (20GB version vs. 8GB version), the percentage of saved TOT and GOT becomes larger. For example, FreqWord-8G has 21% TAT reduction and 40% GOT reduction, while FreqWord-8G reaches

Figure 7: Turnaround time (TAT) and GPU occupancy time (GOT) of three programs on two datasets with GRU off and on.

Figure 8: Cumulative turnaround time and GPU occupancy time savings for opencloud trace dataset.

34% and 49%, respectively. This is intuitive as heavier workloads may exhibit higher degree of redundancy and thus reusability.

5.2.2 Reuse with Different Datasets. In this experiment, we evaluate the case where same functionalities are performed using different datasets. We consider herein a log tracing system on an extremely popular big-data processing framework, Apache Hadoop [24]. We use publicly available data traces of the CMU OpenCloud Hadoop cluster from January of 2011 until June 2011 [24]. We dedicate this experiment to an accumulative data trace analysis that is assumed to be executed monthly. This analysis is run to generate statistical data of each job’s execution time, summed for the year 2011. While jobs rarely span between months, it is nonetheless necessary to include the accumulative results for administrative purposes.

Fig. 8 shows the results of running this scenario for the months of January until June, while we limit the exhibition to three months for clarity purposes. As is evident from the figure, while the database continues to grow (as is shown using connected dots) from 9.8 GB to 13.3 GB, TAT under GRU continues to be consistent, not exhibiting much increase due to the benefit of result reuse.

Another counter-intuitive observation is that GOT dramatically drops after each month. This is because GRU eliminates the need of recomputing the log data from previous months, but rather directly reusing results from previous computations. This trend of reduction in GOT is consistently observed from January until June. Fig. 8 shows the total accumulative GOT savings until the month of June, which yields a 83% reduction.

5.2.3 Reuse with Different Functionalities and Datasets. In this section, we measure the efficacy of GRU under a mixed reuse scenario. A mixed scenario is when both the input data and the computation kernels are different between two runs, as discussed in

Table 2: Reuse of non-identical data & computation.

Warmup	Main Run	TAT (% saved)	GOT (% saved)	Reuse
WS-8G	FW-20G	19.9m (11.56%)	25.2m (17.38%)	4.99%
FW-20G	WS-8G	10.1m (21.09%)	6.4m (45.3%)	12.50%

Sec. 3.2.3. Table 2 shows an example execution under a potential mixed scenario. The first row of Table 2 depicts an example run of FreqWord on Amazon Reviews 20GB (Full). However, prior to its execution, a warmup run of WordStatistics is executed using the Amazon Review 8GB (Compact). As is evident in the table, the reuse possibility under this scenario is noticeable but not significant.

On the other hand, the second row of Table 2 shows a reversed scenario, in which FreqWord on Amazon Reviews 20GB is used as the warmup execution. For the main execution of WordStatistics on Amazon Review 8GB, the savings are quite significant. In the second scenario, the fact that Amazon Review 20GB is an incremental version of Amazon Review 8GB has contributed a significant boost to the execution time of WordStatistics, saving 45% GPU occupancy time in total.

5.3 Experiments with Micro-benchmarks

5.3.1 Setup. The evaluated benchmarks are selected from various application domains and are run in the desktop environment mentioned in Sec. 5.1. These benchmarks include a range applications from the Rodinia test suits [7], CUDA SDK examples [34], a synthesized CUDA game, an object recognition application as listed in Table 3. By increasing the application variety, our goal is to perform a reasonably comprehensive evaluation study and identify GRU’s strengths and limitations when being applied to applications with different characteristics.

The baseline used in our experiments is rCUDA [12], which is a popular API-remoting-based GPU virtualization framework. rCUDA uses the TCP/IP network stack to communicate between its FE and BE. The overhead introduced by the architecture of API-remoting has been extensively discussed in rCUDA [12]. Thus, in our experiments, we mainly compare GRU with rCUDA to evaluate the efficacy of GPU result reuse, since GRU is almost identical to rCUDA if its result reuse component is disabled.

To evaluate the overhead incurred under GRU, we evaluate two settings: (i) we run each experiment under GRU with an empty result cache. Thus in this setting, the measured performance penalty can be viewed as purely the overhead incurred by GRU. We denote this setting by “GRU-miss.” (ii) we run each experiment under GRU assuming that the same experiment has been performed by another VM under GRU (i.e., cached entries have already been established in result cache). Thus, the measured performance in this setting can reflect the effectiveness of GRU. (Note that GRU can only be effective if some reusable results have been cached.) We denote this setting by “GRU-hit”. The main performance metric we adopt in our evaluation is the execution time under rCUDA versus GRU (i.e., from the time when the CUDA context is created to the time when the context is destroyed), as this metric directly reflects the effectiveness of the core GPU result reuse idea implemented.

5.3.2 Results. We have also assessed GRU’s performance and overheads using a set of popular micro-benchmarks, which are shown in Fig. 9. We observe from Fig. 9 that most benchmarks (9

Table 3: Evaluated benchmarks

chess (Chess game AI)	bfs (breadth first search)
lud (LU decomposition)	mmult (matrix multiplication)
nn (nearest neighbor)	bs (BlackScholes modeling)
hs (hotspot simulation)	decode (Image decoding)
hw (medical imaging)	bp (back propagation)
srad (image processing)	hog (Object recognition)

out of 11) reach more than 1.25x speedup compared to rCUDA. Benchmark mmult yields the highest speedup (5x) because of its compute-intensive nature (which can be seen in 10). An important message received from this set of experiments is that GRU is the most effective when applied to compute-intensive applications. This is because the computation time, which is the most time-consuming component of such applications, can be saved by directly reusing the cached results. On the other hand, GRU is also quite effective in handling data-intensive applications. For example, the bfs benchmark, which is data-intensive as seen in Fig. 10, gets a nearly 2.5x speedup. This is mainly because GRU can only optimize the execution of data-intensive applications by merely transferring the hashed value of input data instead of the entire data chunks. Thus, once the cached results associated with the same hashed value are retrieved in LUT, the time-consuming transfer time of input data can be skipped. Moreover, by jointly considering the overhead that GRU introduces when handling these two types of applications, the observation is that GRU performs extremely well for compute-intensive applications due to (i) very small overhead incurred when under GRU-miss, and (ii) large saving of execution time under GRU-hit; while for data-intensive applications, GRU can still achieve a considerable amount of saving of execution time under GRU-hit but at the cost of more overhead under GRU-miss due to large input data hashing.

On the other hand, we observe that one specific kind of application may not benefit a lot from GPU result reuse, *i.e.*, applications that are not compute-intensive, but have rather small input data and large output. This is because in the case of cache-missing, they introduce non-negligible overhead due to inserting the large output in LUT, while in the case of cache-hitting, they cannot gain much performance speedup due to the non-compute-intensive nature. For example, the decode application yields the lowest performance speedup, because the introduced overhead of result insertion and dumping neutralizes the time saving due to the skipped input data transfer and kernel launch time.

We evaluate GRU's runtime overhead compared to rCUDA (which is mainly due to LUT-related operations) by demonstrating the detailed breakdown of applications' execution times on GPU. Fig. 9 depicts the normalized execution time when executing each application under three settings: rCUDA, GRU-miss and GRU-hit. The x-axis of Fig. 9 represents each evaluated application, and the y-axis represents the execution time normalized with respect to the execution time yielded under rCUDA. We observe that the overheads incurred by reuse-miss are less than 5% for four applications, less than 7% for another three applications. Overall, for a majority of the considered benchmarks (7 out of 11), the overhead causes a less than 7% increase in execution times. Among all applications, lud, bfs, and chess yield the greatest overhead (14%-18%). This is

Figure 9: Performance w.r.t. normalized execution time. **Figure 10: The three histograms for each benchmark represent the breakdown under rCUDA, GRU-miss and GRU-hit, respectively.**

because they are either data-intensive or process a large input data, causing more overhead in data hashing, insertion, and movement.

To clearly understand the overhead sources, we have also recorded the detailed breakdown of each benchmark's execution time on GPUs under three scenarios, as shown in Fig. 10. There are in total seven components that contribute to an application's execution time: (i) Init indicates the initialization time of CUDA context, (ii) H2D and D2H indicate data movement between host memory and GPU device memory, (iii) Launch indicates the kernel launch time on GPU, (iv) H+R indicates the time spent on data hashing and probing in LUT, (v) Insert indicates the time spent on inserting results in LUT when caching a result, (vi) Reuse means a kernel launch is skipped by reusing cached results. Through examining the breakdown, we can clearly figure out different characteristics of benchmarks and the overhead differences between GRU-miss and GRU-hit. For the bfs benchmark that incurs the most observable overhead, we can see in Fig. 10 that compared to other benchmarks, the H2D, D2H, and H+R components under bfs contribute a larger portion of the total execution time. The data transfer time in bfs accounts for 81% (67% for H2D and 14% for D2H) of the total execution time under rCUDA, 65% under GRU-miss, and only 20% under GRU-hit. With the delay transfer optimization implemented in GRU, the H2D time of bfs can almost be eliminated under GRU-hit.

6 RELATED WORK

Managing GPUs in the cloud. Current approaches for GPU management in the cloud are classified into I/O pass-through [1], API-remoting [12, 14, 25, 41], para-virtualization [11, 16, 44] and full-virtualization [28, 44, 45, 50], the latter two being two different implementations of the device emulation technique. However, these works do not exploit the idea of GPU computing result reuse.

Computing result reuse. The concept of CPU-based computation reuse has been proposed in the programming language and computer architecture communities. Compiler-assisted approaches [8–10, 42] seek to reuse intermediate results at CPU instruction level. Function-level memoization [31, 37] is used to avoid re-executing functions by caching the results of prior function calls. Moreover, frameworks are proposed to reuse redundant computations at a higher level for the emerging incremental data processing field. For example, Spark [49], Percolator [35], and CBP [27] provide programmers with facilities to store and reuse states across successive runs; while DryadInc [36], Nectar [17], Haloop [6], Incoop [5], CIEL [33], and Shredder [4] are systems that reuse prior computing results. On GPU-incurred reuse, Arnau *et al.* [2] presented a hardware memoization approach to eliminate redundant fragment shader executions on a mobile GPU. Different from these works, GRU focuses on GPGPU and efficiently realizing the partial GPU computing result reuse idea at a GPU kernel launch granularity.

7 CONCLUSION

In this paper, we present GRU, a GPU sharing, result memoization and reuse ecosystem for high performance and cloud computing. GRU exploits computation and data redundancy seen in several important categories of GPU-accelerated workloads. We have fully implemented GRU and evaluation results show that GRU is effective in improving the turnaround time and reducing the GPU occupation time, while incurring a rather small runtime overhead.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable feedback and suggestions that helped improve the paper. The work is supported by the National Science Foundation (NSF) through CNS 1527727 and CNS CAREER 1750263.

REFERENCES

- [1] AMAZON. 2006. Amazon Elastic Compute Cloud (Amazon EC2). (2006). <http://aws.amazon.com/ec2>
- [2] Jose-Maria Arnau, Joan-Manuel Parcerisa, and Polychronis Xekalakis. 2014. Eliminating redundant fragment shader executions on a mobile GPU via hardware memoization. In *ISCA'14*.
- [3] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the Art of Virtualization. In *SOSP'03*.
- [4] Pramod Bhatotia, Rodrigo Rodrigues, and Akshat Verma. 2012. Shredder: GPU-accelerated incremental storage and computation. In *FAST'12*.
- [5] Pramod Bhatotia, Alexander Wieder, Rodrigo Rodrigues, Umut A Acar, and Rafael Pasquin. 2011. Incoop: MapReduce for incremental computations. In *SOC'11*.
- [6] Yingyi Bu, Bill Howe, Magdalena Balazinska, and Michael D Ernst. 2010. Haloop: efficient iterative data processing on large clusters. *Proc. VLDB Endow* (2010).
- [7] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W Sheaffer, Sang-Ha Lee, and Kevin Skadron. 2009. Rodinia: A benchmark suite for heterogeneous computing. In *IISWC'09*.
- [8] DA Connors and W-MW Hwu. 1999. Compiler-directed dynamic computation reuse: rationale and initial results. In *MICRO'99*.
- [9] Daniel A. Connors, Hillery C. Hunter, Ben-Chung Cheng, and Wen-mei W. Hwu. 2000. Hardware Support for Dynamic Activation of Compiler-directed Computation Reuse. In *ASPLOS'00*.
- [10] Yonghua Ding and Zhiyuan Li. 2004. A compiler scheme for reusing intermediate computation results. In *CGO'04*.
- [11] Micah Dowty and Jeremy Sugerman. 2009. GPU virtualization on VMware's hosted I/O architecture. *SIGOPS Oper. Syst. Rev.* (2009).
- [12] José Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Orti. 2010. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPSC'10*.
- [13] Ahmad Ghazal, Tilmann Rabl, Mingqing Hu, Francois Raab, Meikel Poess, Alain Crolette, and Hans-Arno Jacobsen. 2013. BigBench: Towards an Industry Standard Benchmark for Big Data Analytics. In *SIGMOD'13*.
- [14] Giulio Giunta, Raffaele Montella, Giuseppe Agrillo, and Giuseppe Coviello. 2010. A GPGPU transparent virtualization component for high performance computing clouds. In *Euro-Par'10*.
- [15] Google. 2011. Google Cloud Platform. (2011). <https://cloud.google.com>
- [16] Mathias Gottschlag, Marius Hillenbrand, Jens Kehne, Jan Stoess, and Frank Bellosa. 2013. LoGV: Low-overhead GPGPU virtualization. In *FHC'13*.
- [17] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A Thekkath, Yuan Yu, and Li Zhuang. 2010. Nectar: Automatic Management of Data and Computation in Datacenters. In *OSDI'10*.
- [18] F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. *ACM Trans. Interact. Intell. Syst.* (2015).
- [19] D. Richard Hipp. 2018. SQLite3 In-Memory Databases. <https://www.sqlite.org/inmemorydb.html>. (2018).
- [20] S. Huang, J. Huang, J. Dai, T. Xie, and B. Huang. 2010. The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *ICDEW'10*.
- [21] IBM. 2014. IBM Bluemix Infrastructure. (2014). <https://www.ibm.com/bluemix>
- [22] IBM. 2016. IBMSparkGPU. <https://github.com/IBMSparkGPU/SparkGPU>. (2016).
- [23] Y. Koren, R. Bell, and C. Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* (2009).
- [24] Parallel Data Lab. 2015. OpenCloud Hadoop cluster trace. (2015). <http://ftp.pdl.cmu.edu/pub/datasets/hla/dataset.html>
- [25] H Andrés Lagar-Cavilla, Niraj Tolia, Mahadev Satyanarayanan, and Eyal De Lara. 2007. VMM-independent graphics acceleration. In *VEE*.
- [26] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO'04*.
- [27] Dionysios Logothetis, Christopher Olston, Benjamin Reed, Kevin C Webb, and Ken Yocum. 2010. Stateful bulk processing for incremental analytics. In *SOC'10*.
- [28] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafir. 2015. rIOMMU: Efficient IOMMU for I/O devices that employ ring buffers. In *ASPLOS'15*.
- [29] Julian McAuley, Christopher Targett, Qinfeng Shi, and Anton van den Hengel. 2015. Image-Based Recommendations on Styles and Substitutes. In *SIGIR'15*.
- [30] Michael D McCool, Arch D Robison, and James Reinders. 2012. *Structured parallel programming: patterns for efficient computation*. Elsevier.
- [31] Donald Michie. 1968. Memo functions and machine learning. *Nature* (1968).
- [32] Microsoft. 2010. Microsoft Azure: Cloud Computing Platform & Services. (2010). <https://azure.microsoft.com>
- [33] Derek Gordon Murray, Malte Schwarzkopf, Christopher Smowton, Steven Smith, Anil Madhavapeddy, and Steven Hand. 2011. CIEL: A Universal Execution Engine for Distributed Data-Flow Computing. In *NSDI'11*.
- [34] NVIDIA. 2007. Compute unified device architecture programming guide. (2007).
- [35] Daniel Peng and Frank Dabek. 2010. Large-scale Incremental Processing Using Distributed Transactions and Notifications. In *OSDI'10*.
- [36] Lucian Popa, Mihai Budiu, Yuan Yu, and Michael Isard. 2009. DryadInc: Reusing Work in Large-scale Computations. In *HotCloud'09*.
- [37] William Pugh and Tim Teitelbaum. 1989. Incremental computation via function caching. In *POPL'89*.
- [38] Christopher J Rossbach, Jon Currey, Mark Silberstein, Baishakhi Ray, and Emmett Witchel. 2011. PTask: Operating system abstractions to manage GPUs as compute devices. In *SOSP'11*.
- [39] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. 2013. Dandelion: a compiler and runtime for heterogeneous systems. In *SOSP'13*.
- [40] Mehrzad Samadi, Davoud Anoushe Jamshidi, Janghaeng Lee, and Scott Mahlke. 2014. Paraprox: Pattern-based approximation for data parallel applications. In *ASPLOS'14*.
- [41] Lin Shi, Hao Chen, Jianhua Sun, and Kenli Li. 2012. vCUDA: GPU-accelerated high-performance computing in virtual machines. *IEEE Trans. Comput.* (2012).
- [42] Avinash Sodani and Gurindar S Sohi. 1997. Dynamic instruction reuse. In *ISCA'97*.
- [43] STP. 2014. The Simple Theorem Prover. (2014). <http://stp.github.io/>
- [44] Yusuke Suzuki, Shinpei Kato, Hiroshi Yamada, and Kenji Kono. 2014. GPUvm: Why Not Virtualizing GPUs at the Hypervisor. In *ATC'14*.
- [45] Kun Tian, Yaozu Dong, and David Cowperthwaite. 2014. A full GPU virtualization solution with mediated pass-through. In *ATC'14*.
- [46] L. Wang, J. Zhan, C. Luo, Y. Zhu, Q. Yang, Y. He, W. Gao, Z. Jia, Y. Shi, S. Zhang, C. Zheng, G. Lu, K. Zhan, X. Li, and B. Qiu. 2014. BigDataBench: A big data benchmark suite from internet services. In *HPCA'14*.
- [47] Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Rounne, Rob Springer, Xuetao Weng, and Robert Hundt. 2016. gpuc: an open-source GPGPU compiler. In *CGO'16*.
- [48] Y.C. 2012. Extremely fast non-cryptographic hash algorithm. <https://code.google.com/archive/p/xxhash>. (2012).
- [49] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, and Ion Stoica. 2010. Spark: cluster computing with working sets. In *HotCloud'10*.
- [50] Husheng Zhou, Yangchun Fu, and Cong Liu. 2015. Supporting Dynamic GPU Computing Result Reuse in the Cloud. In *HotCloud*.