Noname manuscript No.
(will be inserted by the editor)

Out-of-core Implementation for Accelerator Kernels on Heterogeneous Clouds

Hamidreza Khaleghzadeh · Ziming Zhong · Ravi Reddy · and Alexey Lastovetsky

the date of receipt and acceptance should be inserted later

Abstract Cloud environments today are increasingly featuring hybrid nodes containing multicore CPU processors and a diverse mix of accelerators such as Graphics Processing Units (GPUs), Intel Xeon Phi co-processors, and Field-Programmable Gate Arrays (FPGAs) to facilitate easier migration to them of HPC workloads. While virtualization of accelerators in clouds is a leading research challenge, we address the programming challenges that assail execution of large instances of data-parallel applications using these accelerators in this paper.

In a typical hybrid node in a cloud, the tight integration of accelerators with multicore CPUs via PCI-E communication links contains inherent limitations such as limited main memory of accelerators and limited bandwidth of the PCI-E communication links. These limitations poses formidable programming challenges to execution of large problem sizes on these accelerators. In this paper, we describe a library containing interfaces (*HCLOOC*) that addresses these challenges. It employs optimal software pipelines to overlap data transfers between host CPU and the accelerator and computations on the accelerator. It is designed using the fundamental building blocks, which are OpenCL command queues for FPGAs, Intel offload streams for Intel Xeon Phis, and CUDA streams and events that allow concurrent utilization of the copy and execution engines provided in NVidia GPUs.

We elucidate the key features of our library using an out-of-core implementation of matrix multiplication of large dense matrices on a hybrid node, an Intel Haswell multicore CPU server hosting three accelerators that includes

H. Khaleghzadeh · R. Reddy · A. Lastovetsky
School of Computer Science, University College Dublin, Belfield, Dublin 4, Ireland,
E-mail: hamidreza.khaleghzadeh@ucdconnect.ie, ravi.manumachu@ucd.ie, alexey.lastovetsky@ucd.ie

Z. Zhong
Complex Aviation Systems Simulation Laboratory, Beijing, China,
E-mail: zhongziming@gmail.com

NVidia K40c GPU, Intel Xeon Phi 3120P, and a Xilinx FPGA. Based on experiments with the GPU, we show that our out-of-core implementation achieves 82% of peak double-precision floating performance of the GPU and a speedup of 2.7 times over the NVidia's out-of-core matrix multiplication implementation (CUBLAS-XT). We also demonstrate that our implementation exhibits 0% drop in performance when the problem size exceeds the main memory of the GPU. We observe this 0% drop also for our implementation for Intel Xeon Phi and Xilinx FPGA.

Keywords Heterogeneous clouds, GPU, Intel Xeon Phi, matrix multiplication, out-of-core, CUBLAS, CUDA, Intel MKL

1 Introduction

Cloud computing systems today are placing great emphasis in facilitating easier migration and execution of HPC workloads by trying to solve several daunting challenges that have impeded this process. The challenges include: a) Performance variation of the workloads due to virtualization, b) Poor network performance due to wide geographical separation of the computational resources, and c) Provision for a diverse mix of hardware accelerators, which have now become dominant in the HPC landscape.

While reduction or even complete removal of performance losses due to virtualization remains the biggest obstacle to overcome, commercial clouds have addressed other easily surmountable challenges that allowed HPC users to think about migrating their applications to cloud or using the HPC resources in the cloud at a relatively low cost. They now provide reservation of hybrid nodes that contain a mix of high performance processors now ubiquitous in HPC such as Graphics Processing Units (GPUs), Intel Xeon Phi co-processors (PHIs), and Field-Programmable Gate Arrays (FPGAs).

Filelis-Papadopoulos et al. [1] study few HPC applications that have been migrated to clouds, which include Oil and Gas exploration, Genomics, and Ray-tracing. They present a characterization of hardware used in these three applications. Lynn et al. [2] study the conception of heterogeneous clouds that use machines of different types, which include modern high performance energy-efficient multicore processors and accelerators. They propose a cloud management and delivery architecture that endeavours to enable cloud services for HPC.

Hardware acceleration of scientific kernels in cloud poses three prominent challenges. First, well-known hardware accelerators such as GPUs, PHIs, and FPGAs must be provided and allowed to be booked with less hassle by HPC users for execution of their applications. Second, virtualization of hardware accelerators must be made possible to meet the core principles of the cloud model, which includes elasticity and on-demand computing. Finally, efficient utilization of accelerators to solve big instances of data-parallel applications must be facilitated. The first challenge has now been addressed effectively by

Table 1 Specification of the Intel Haswell multicore CPU server.

Technical Specifications	Intel Haswell Server
Processor	Intel E5-2670 v3 @ 2.30GHz
OS	CentOS 7
Microarchitecture	Haswell
Memory	64 GB
Socket(s)	2
Core(s) per socket	12
NUMA node(s)	2
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	30720 KB
TDP	240 W
Base Power	58 W

Table 2 Main specifications of NVidia K40c GPU.

Characteristic	Description
CUDA Driver Version / Runtime Version	7.5 / 7.5
CUDA Capability Major/Minor version number	3.5
Total amount of global memory	11520 MBytes
(15) Multiprocessors, (192) CUDA Cores/MP	2880 CUDA Cores
L2 Cache Size	1572864 bytes
Total amount of constant memory	65536 bytes
Total amount of shared memory per block	49152 bytes
Total number of registers available per block	65536
Warp size	32
Concurrent copy and kernel execution	Yes with 2 copy engine(s)
Support host page-locked memory mapping	Yes
Idle power consumption (Watts)	60

the cloud service providers. Virtualization of hardware accelerators at zero performance loss remains a formidable research challenge. Hong et al. [3] present a GPU virtualization system that achieves maximum fairness while causing little performance degradation. In this paper, we address the final challenge.

Integration of multicore CPUs with accelerators poses challenges to execution of large problem sizes on these accelerators. The challenges are:

1. **Limited memory size.** Accelerators typically have smaller main memory than the host multicore CPU. Therefore, the maximum problem size that can be solved by an accelerator is limited by its main memory size. Consider the Intel Haswell multicore CPU server shown in Table 1 hosting a NVidia GPU shown in Table 2 and an Intel Xeon Phi co-processor in Table 3. The server includes two Intel Xeon E5-2670 CPUs. While the host CPU

Table 3 Specification of the Intel Xeon Phi 3120P.

Technical Specifications	**Intel Xeon Phi 3120P**
No. of processor cores	57
Base frequency	1.10 GHz
Total main memory	6 GB GDDR5
L2 cache size	28.5 MB
Memory bandwidth	240 GB/sec
Memory clock	3.0 GHz
TDP	300 W
Idle Power	91 W

Table 4 Specification of the Xilinx Virtex 7 690T FPGA.

Technical Specifications	**Xilinx Virtex 7 690T FPGA**
Frequency	200 MHz
LUTs	693120
DSPs	3600
BRAM	53 MB
FFs	866,400
Total main memory	16 GB DDR3

contains 64GB main memory, NVidia GPU has only 12 GB main memory and Intel Xeon Phi has only 6 GB main memory. Therefore, to execute large problem sizes of an application using these accelerators, it is required that out-of-core implementations of the application are either available or developed from scratch.

2. **Limited bandwidth of the PCI-E communication link.** Out-of-core executions usually entail multiple data transfers of data structures (that fit inside the main memory of the accelerator) from the host CPU to the accelerator and back. Accelerators are connected to CPUs using PCI-E communication links. However, due to the limited bandwidth of the PCI-E communication link, this impacts the execution times of the out-of-core implementation. In our server, the NVidia GPU and Intel Xeon Phi communicate with the host CPU using low-bandwidth PCI-E x16 links. The FPGA communicates using PCI-E x8 link. To achieve maximum performance, an out-of-core implementation must utilize the vendor-supplied optimizations for data transfers to overlap the communications over the PCI-E communication link with the computations on the accelerator.
3. **Library for Out-of-core Implementations.** There is an abysmal lack of libraries providing interfaces that allow programmers to write out-of-core implementations for their data-parallel kernels on accelerators. There are exceptions (but very few) such as NVidia's CUBLAS-XT package [4], which provides a set of BLAS routines that utilize multiple GPUs and MAGMA [5], which provides out-of-core dense matrix factorizations. However, from our experiments, it is observed that vendor out-of-core implementations (such as [4]) are not the best in terms of performance.

In this paper, we present a library (*HCLOOC*) that allows programmers to write out-of-core implementations of data-parallel kernels for accelerators such as GPUs, PHIs, and FPGAs. The library is a wrapper that reuses the fundamental building blocks such as OpenCL command queues [6] for FPGAs, Intel offload streams [7] for Intel Xeon Phis, and CUDA streams and events that allow concurrent utilization of the copy and execution engines provided in NVidia GPUs [8], [9].

The library contains two principal components. The first component, *Partitioner*, partitions the workload into blocks where each block can fit into the accelerator's main memory. The second component, *Stream Engine*, uses a configurable software pipeline to overlap data transfers from host CPU to the accelerator and back and kernel invocations in the accelerator. This component reuses the vendor-supplied optimization engines for the data transfers. For example, for the out-of-core implementation of matrix multiplication that we present in this paper, the *Stream Engine* uses a five-stage pipeline.

In this paper, we present the essence of the principal components of the library. The library interface design contained creation of a uniform interface for the fundamental building blocks, OpenCL command queues [6] for FPGAs, Intel offload streams [7] for Intel Xeon Phis, and CUDA streams and events [8], which have disparate interfaces. This turned out to be a considerably difficult task, which we aim to present in our future work.

We elucidate the key features of our library by implementing out-of-core execution of matrix multiplication of large dense matrices on a hybrid node, which contains three accelerators including NVidia K40c GPU, Intel Xeon Phi 3120P, and a Xilinx FPGA. Based on experiments on an Intel Haswell multicore CPU server hosting a NVidia K40c GPU, we show that our out-of-core implementation achieves 82% of the peak double-precision floating performance of the GPU and a speedup of 2.7 times over the NVidia's CUBLAS-XT out-of-core DGEMM solver. We also demonstrate that our implementation exhibits 0% drop in performance when the problem size exceeds the main memory of the GPU. We observe this 0% drop also of our implementations for Intel Xeon Phi and Xilinx FPGA.

Our main contributions can be summarized as follows:

- A library (*HCLOOC*) that allows programmers to write out-of-core implementations of data-parallel kernels for accelerators such as GPUs, Xeon Phis, and FPGAs. The key contributions of the library are:
  - Interfaces that allow programmers to employ configurable software pipelines to overlap data transfers from host CPU to the accelerator and back and invocations of in-core kernels in the accelerator.
  - A uniform interface for the fundamental building blocks, OpenCL command queues [6] for FPGAs, Intel offload streams [7] for Intel Xeon Phis, and CUDA streams and events [8], which have disparate interfaces.
- An efficient out-of-core implementation written using *HCLOOC* of matrix multiplication for NVidia GPU that outperforms the NVidia's out-of-core

implementation ([4]) and that exhibits 0% drop in performance when problem sizes exceed the main memory of the accelerator.
- An efficient out-of-core implementation written using *HCLOOC* of matrix multiplication for Intel Xeon Phi that exhibits 0% drop in performance when problem sizes exceed the main memory of the accelerator.
- The very first out-of-core implementation of matrix multiplication for Xilinx FPGA that also exhibits 0% drop in performance when problem sizes exceed the main memory of the accelerator.

The paper is organized as follows. Section 2 contains related work on HPC in cloud, hardware acceleration in cloud, and research works focusing on out-of-core implementations for accelerators. Section 3 presents *HCLOOC*. Section 3.1 describes out-of-core implementation of matrix multiplication using the core features of the library. Section 4 presents experimental results. Finally, section 5 concludes the paper.

2 Related Work

Our survey is divided into three parts. In the first part, we review prominent works that present efforts to facilitate execution of HPC in a cloud environment and also works that highlight the adverse effect on performance introduced by virtualisation. In the second part, we review notable works that present virtualization of hardware acceleration in commercial clouds. In the final part, we focus entirely on efforts presenting out-of-core implementations for accelerator kernels.

2.1 HPC in Cloud Environments

Ostermann et al. [10] study the performance and reliability of a production cloud and find them to be low. Iosup et al. [11] analyze the performance of cloud computing services for scientific computing workloads for four commercial clouds and conclude that current clouds need a performance improvement by an order of magnitude in order to be useful to the scientific community.

Gupta et al. [12] present a HPC-aware scheduler of virtual instances that aim to lessen the impact of cross-VM interference on the performance. Parashar et al. [13] look at usage modes for HPC in cloud. They propose a concept of *HPC as a Service* that provides bare-metal performance that is usually required for HPC applications.

Mauch et al. [14] present an overview of high performance cloud computing efforts and virtualization techniques. They propose a new model for HPC Infrastructure as a Service (IaaS).

2.2 Hardware Acceleration in Cloud Environments

Giunta et al. [15] propose a GPU virtualization and sharing service that enables a virtual machine instance to access GPUs in a transparent and hypervisor independent way with a moderate overhead over real GPU. Byma et al. [16] present an approach for integrating virtualized FPGA-based hardware accelerators into commercial cloud environments with minimal virtualization overhead. Hong et al. [17] surveyed GPU virtualization techniques and scheduling methods. They considered virtualization techniques implemented at the GPU library, driver, and hardware levels. In addition, the survey involves GPU scheduling methods that address performance and fairness issues between multiple virtual machines sharing GPUs.

2.3 Out-of-core Implementation of Accelerator Kernels

Gu et al. [18] present an out-of-core implementation of FFT kernel for a single GPU. The authors co-optimized both CPU-GPU data transfer via PCI-E bus and on-GPU computation for 1D, 2D and 3D FFTs by using the Cooley-Tukey decomposition framework. The framework is used for decomposing a large sized FFT into smaller sub-FFTs, which are then transferred to the GPU in batches. A recursive kernel is proposed to compute on-card FFT. To achieve high throughput on the CPU-GPU data channel, a blocked buffer technique for 1D FFTs was developed. The effect of sub-array size on data transfer performance is also studied in this paper. They find that PCI-E bus bandwidth decreases when sub-array size decreases due to the consequent increase in the number of *cudaMemcpyAsync* calls. To deal with small sub-arrays and to increase the PCI-E bus bandwidth, the buffering of continuous sub-arrays and the transfer of all of them using a single *cudaMemcpyAsync* call are proposed. Mu et al. [19] introduced an out-of-core algorithm for LU decomposition. The proposed approach is based on the left-looking factorization on GPU/CPU platform where it uses both the host memory and the hard disk for out-of-core computations.

In 2012, Ziming et al. [20],[21] proposed an out-of-core implementation for matrix multiplication routine (DGEMM) for NVidia GPU. However, the implementation placed some constraints on the dimensions of the matrices that are allowed in the matrix multiplication. In this work, we removed these constraints and applied additional optimizations to improve the performance of our proposed out-of-core library. In 2016, Wu et al. [22] presented an out-of-core dense matrix multiplication implementation for CPU-GPU platforms similar to [20],[21]. They performs matrix decomposition according to peak bandwidth of PCI-E links and the bandwidth required by application.

The CUBLAS-XT library [4] provides a set of BLAS routines that utilize multiple GPUs connected to the same motherboard. It uses CUDA streams [8] and events to efficiently manage data transfers across PCI-Express bus and kernel invocations on the GPUs. The routines in the library also support out-

of-core operation where the size of the matrices are limited only by the system memory size. However, we show in this work that our out-of-core implementation of the DGEMM routine out-performs that provided in CUBLAS-XT library. SciGPU-GEMM [23] is a library of wrapper functions to help use the GEMM routines from CUBLAS on GPUs with limited memory and no double precision hardware. HPL-CUDA [24] is a library for high performance computing Linpack benchmark for CUDA. It does not contain out-of-core implementation for level 3 BLAS matrix multiplication routine.

3 Out-of-core Library for Accelerator Kernels (*HCLOOC*)

In this section, we present a library, called *HCLOOC*, which allows programmers to write out-of-core implementations for their kernels on accelerators such as GPUs, PHIs, and FPGAs.

HCLOOC consists of two principal components: *Partitioner* and *Stream Engine*.

Partitioner contains interfaces, which allow partitioning of input and output data structures into partitions that fit into an accelerator's main memory.

Stream Engine uses a configurable software pipeline to overlap data transfers from host CPU to the accelerator and back and invocations of in-core kernels in the accelerator. It is a wrapper that utilizes the fundamental building blocks such as OpenCL command queues [6] for FPGAs, Intel offload streams [7] for Intel Xeon Phis, and CUDA streams that allow concurrent utilization of the copy and execution engines provided in NVidia GPUs [8], [9]. For example, for the out-of-core implementation of matrix multiplication that we present in this paper, the *Stream Engine* uses a five-stage pipeline.

3.1 Implementation for Dense Matrix Multiplication on a GPU using *HCLOOC*

In this section, we elucidate the core logic in the two components (*Partitioner* and *Stream Engine*) by describing our out-of-core implementation of matrix multiplication of large dense matrices on NVidia GPUs.

The implementation computes $C = \alpha \times A \times B + \beta \times C$, where A, B, and C are matrices of size $M \times K$, $K \times N$, and $M \times N$, respectively and α and β are constant floating-point numbers. If workload size ($M \times K + K \times N + M \times N$) fits into the memory of GPU, all three matrices are transferred to the device, the kernel CUBLAS DGEMM [25] is then invoked to update matrix C, and the resultant matrix C is returned to the host. But, when the workload size exceeds the main memory of the accelerator, the data transfer of the matrices will fail.

First step in our out-of-core implementation is partitioning of matrices A, B, and C. *Partitioner* splits matrix A into h equal horizontal slices, matrix B into v equal vertical slices, and matrix C into $h \times v$ equal rectangular blocks

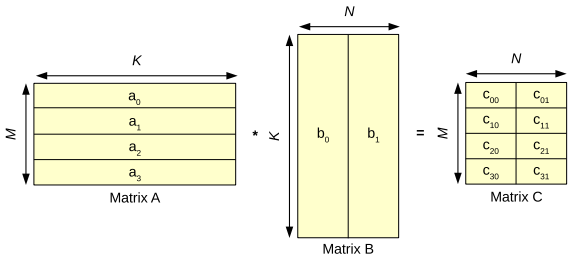

Fig. 1 Using *Partitioner* module for decomposition of matrix A into 4 horizontal slices, matrix B into 2 vertical slices, and matrix C into 8 ($= 4 \times 2$) blocks.

ensuring that the data required for updating of any two blocks of C in the same column is small enough to fit in the accelerator's memory.

For example, suppose M, N and K to be 4, 4 and 8 respectively, resulting in the total workload size equal to 80 ($4 \times 8 + 8 \times 4 + 4 \times 4$) matrix elements. Suppose the GPU's main memory can only store 44 matrix elements. Then *Partitioner* will be applied, and it will split matrix A into 4 horizontal slices, matrix B into 2 vertical slices and consequently matrix C into eight 1×2 blocks guaranteeing that the data required for updating of any two blocks of C in the same column will fit in the memory of the accelerator. Figure 1 shows the matrix decomposition. Although other decompositions are possible (for example, partitioning A, B and C into 2, 4 and 8 sub-blocks respectively), *Partitioner* will return the decomposition, which additionally optimizes the work of the target software pipeline. In this particular case, the pipeline uses two sets of buffers and two parallel streams, and in order to optimize the use of the resources, *Partitioner* is instructed to select the decomposition with the smallest possible v.

Stream Engine is then employed to execute the out-of-core implementation. CUDA streams and asynchronous communications are used to optimally utilize concurrent access of copy and execution engines provided in NVidia GPUs thereby achieving optimal overlapping of communication with computation.

The columns of blocks C are computed one after the other. In each column, the blocks are computed going from the top to the bottom. Each iteration is associated with multiple transfers of the matrix blocks between the host memory and device memory, which leads to a significant communication cost. To reduce the communication overhead, *HCLOOC* overlaps data transfers and kernel invocations. To achieve this, two sets of data buffers are allocated in the GPU's main memory. Each set is used for updating one block of C. While one block of C is being updated, the required data for the second block of C is transferred into the second set of buffers.

Stream Engine uses a five-stage software pipeline to execute the out-of-core implementation. Figure 2 presents the pipeline structure for three matrices A, B and C decomposed in the Figure 1. The stages of pipeline are described as following:

– **S(b_i):** Sending a i_th slice of matrix B (b_i) from host to device.

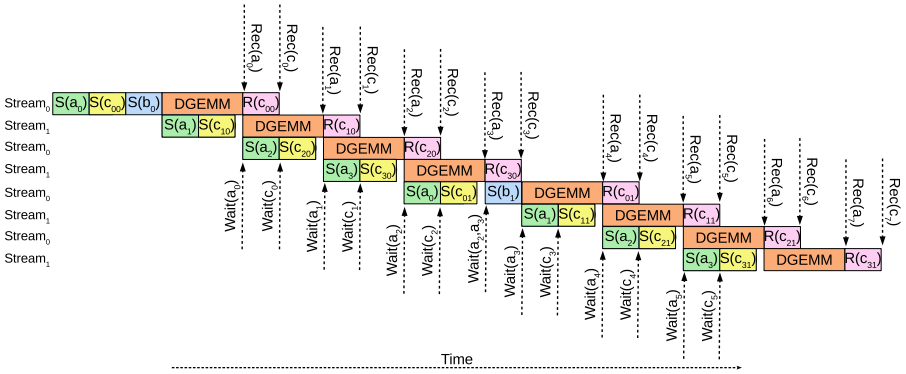

Fig. 2 Pipeline structure in *Stream Engine* module for sample matrices shown in Figure 1 on a GPU with dual copy engines and one execution engine which supports concurrent data transfers in two directions (represented by $S()$ calls) and overlapping of data transfers and kernel executions (represented as DGEMM). Events, $Rec(x)$ and $Wait(x)$, are used for synchronization of data transfers.

- **S(a_i):** Sending a i_th slice of matrix A (a_i) from host to device.
- **S(c_{ij}):** Sending a rectangular block of matrix C (c_{ij}) from host to device.
- **DGEMM:** Vendor-supplied optimized DGEMM (CUBLAS) invocation.
- **R(c_{ij}):** Sending the updated block c_{ij} of C back from device to host.

Since blocks on matrix C are updated in the column order, the first stage of pipeline (S(b_i)) occurs every h step (h is the number of horizontal slices). The stream, which updates c_{ij}, transfers horizontal slice a_i, vertical slice b_j (if it has not already been transferred into the accelerator memory), and block c_{ij} from host to the accelerator memory. After updating c_{ij} by invoking in-core CUBLAS DGEMM, it is sent back to the host. In figure 2, it is supposed that GPU is provided with dual copy engines, which supports concurrent data transfers in two directions. Since creating a new stream has some overhead, we exploit and reuse just two streams in a round robin order so that while one stream is involved in doing computation, the other is transferring.

To make sure data stored in device buffers will not be overwritten until kernel executions that operate on the data have completed, we create events for each sub-matrix existing in A and C. As shown in figure 2, $Rec(x)$ represents recording the event associated with block x, and $Wait(x)$ makes the process wait for the event associated with block x until it is recorded.

We have explained *HCLOOC* using dense matrix multiplication which is one of level 3 BLAS routines. To implement out-of-core matrix-vector operations (level 2 BLAS routines) using *HCLOOC*, matrix A is partitioned. Vector b is completely transferred to the device and stored from start to completion of the out-of-core operation. Vector c is also partitioned and updated during the course of the out-of-core operation. In our future work, we will consider triangular and banded matrices.

3.1.1 Stream Engine*: Further Details*

Stream Engine is responsible for transferring input data from host CPU to GPU, invocations of in-core *CUBLAS_DGEMM* [25], an implementation of BLAS (Basic Linear Algebra Subprograms) on top of the NVidia CUDA runtime, and transferring the resultant output blocks back to the host.

Since out-of-core computation is associated with lots of data transfer between host CPU and the accelerator, we use two sets of buffers on the GPU which include 5 buffers. Two out of 5 buffers, $dA[0]$ and $dA[1]$, store two slices of A, one buffer, dB, stores one slice of B and the remaining two buffers, $dC[0]$ and $dC[1]$, are used for sub-blocks of C. Employing two sets of buffers along with two CUDA streams enables communication-computation overlapping on the accelerator.

Algorithm 1 illustrates the work of *Stream Engine* for NVidia GPUs. Inputs to the module are matrices A, B and C, with sizes of $M \times K$, $K \times N$ and $M \times N$ respectively, and h and v which are determined by *Partitioner* (section 3.1.2). There exist two CUDA streams (Line 2). Operations issued into a stream are executed in issue-order, while operations submitted to different streams can be executed concurrently. Since the GPU supports concurrent copy and execution engines, the designed out-of-core matrix multiplication implementation utilizes concurrent data transfers in both directions. For synchronization of data communications, we create two sets of CUDA event arrays (Line 3).

At the beginning, slices a_0, b_0 and block $c_{0,0}$ are transferred into device buffers $dA[0]$, dB and $dC[0]$ (Lines 9-13). While $dC[set]$, $set = \{0, 1\}$, is being updated by in-core CUBLAS_DGEMM kernel (line 15), next sub-matrices of A, C (Lines 17-22) and following sub-matrix of B (if it is applicable) (Lines 23-27) are asynchronously transferred to the device by the other stream. After finishing its computation, the current stream records $event_a$ to release buffer dA (Line 16). Line 28 is responsible for sending the result back to the host. Then, the current stream releases its dC (Line 29) to be reused by the other stream. Finally, the last block, $c_{(h-1)(j-1)}$, is updated and sent back to the host memory (Lines 31, 32).

The amount of communication that could be overlapped with the computation depends on the ratio of the communication time and the computation time. If the CUBLAS_DGEMM execution dominates the total execution time, then, the smaller the ratio, the more communication could be overlapped. However, if the communication over the PCI-E bus dominates the total execution time, the communication would always be a bottleneck.

We have elucidated the principal components of *HCLOOC* by implementing out-of-core dense matrix multiplication for NVidia GPUs. For out-of-core implementations for PHIs and FPGAs, *Stream Engine* uses Intel offload streams (for PHIs) and OpenCL command queues (for FPGAs). Computational kernels for PHIs and FPGAs would be vendor-optimized BLAS library routine DGEMM.

Algorithm 1 *Stream Engine* using CUDA streams and events to execute out-of-core DGEMM implementation

```
1: function Stream Engine(A, B, C, M, N, K, h, v)
2:     Stream stream[2]
3:     Event event_a[h * v], event_c[h * v]
4:     for j = 0; j < v; j++ do
5:         for i = 0; i < h; i++ do
6:             idx ← i + j * h
7:             set ← idx % 2 , set_ ← (idx + 1) % 2
8:             i_ ← (idx + 1) % h , j_ ← (idx+1)/h
9:             if idx = 0 then
10:                MEMCPYASYNC(b_0 -> dB, stream[idx%2])
11:                MEMCPYASYNC(a_0 -> dA[set], stream[idx%2])
12:                MEMCPYASYNC(c_{0,0} -> dC[set], stream[idx%2])
13:            end if
14:            if idx < (h * v − 1) then
15:                CUBLAS_DGEMM(dA[set], dB, dC[set], stream[idx%2])
16:                EVENTRECORD(event_a[idx], stream[idx%2])
17:                if idx > 0 then
18:                    STREAMWAITEVENT(stream[(idx + 1)%2], event_a[idx − 1])
19:                    MEMCPYASYNC(a_{i_} -> dA[set_], stream[(idx + 1)%2])
20:                    STREAMWAITEVENT(stream[(idx + 1)%2], event_c[idx − 1])
21:                    MEMCPYASYNC(c_{i_,j_} -> dC[set_], stream[(idx + 1)%4]))
22:                end if
23:                if i = (h − 1) AND j < (v − 1) then
24:                    STREAMWAITEVENT(stream[(idx + 1)%2], event_a[idx])
25:                    STREAMWAITEVENT(stream[(idx + 1)%2], event_a[idx − 1])
26:                    MEMCPYASYNC(b_{j+1} -> dB, stream[(idx + 1)%2]))
27:                end if
28:                MEMCPYASYNC(dC[set] -> c_{i,j}, stream[idx%2])
29:                EVENTRECORD(event_c[idx], stream[idx%2])
30:            else
31:                CUBLAS_DGEMM(dA[set], dB, dC[set], stream[idx%2])
32:                MEMCPYASYNC(dC[set] -> c_{i,j}, stream[idx%2])
33:            end if
34:        end for
35:    end for
36: end function
```

3.1.2 Partitioner*: Further Details*

Partitioner decomposes matrix A into h horizontal slices, a_i $(0 \leq i \leq h-1)$, B into v vertical slices, b_j $(0 \leq j \leq v-1)$, and matrix C consequently into $h * v$ blocks, $c_{i,j}$. Partitioning of the matrices is performed such that certain constraints and optimization criteria are satisfied:

- Every matrix is decomposed into sub-matrices of approximately the same size. This ensures load balancing and maximum concurrency.
- GPU's main memory is divided between 5 buffers organized into two sets: $dA[0]$ and $dC[0]$ in one set, $dA[1]$ and $dC[1]$ in the other set. dB is shared between the sets.
- The number of slices in matrix B should be as small as possible. There is only one buffer on the accelerator for matrix B. Since the buffer is

shared between two streams, data transfer from CPU to GPU cannot be overlapped with computation for matrix B slices, and this degrades the communication-computation overlap in the pipeline structure. Therefore, *Partitioner* decomposes matrices so to minimize the number of slices of B.
- Minimizing the number of slices in B may result in too many slices in A. We have experimentally found that the performance of *HCLOOC* degrades when matrix A is partitioned into too many slices. To prevent this, *Partitioner* minimizes the product: $h \times v$.

Algorithm 2 shows the core logic of *Partitioner* in this specific case where the inputs are matrix sizes, M, N and K, and the memory size of the accelerator is mem_size. Outputs are h, v, $heights$ and $widths$. $heights$ is an array of size h where $heights[i]$ contains the number of rows in the i-th slice of matrix A. Similarly, $widths$ is an array of size v where $widths[i]$ contains the number of columns in the i-th slice of matrix B.

We know that the size of $dA[set]$ is $\frac{M}{h} \times K$, dB is $K \times \frac{N}{v}$, and $dC[set]$ is $\frac{M}{h} \times \frac{N}{v}$, where $set = \{0, 1\}$. Since all 5 buffers should fit into the accelerator memory, buffer sizes must satisfy the following equation (1).

$$2 \times \frac{M}{h} \times K + K \times \frac{N}{v} + 2 \times \frac{M}{h} \times \frac{N}{v} = mem\_size \tag{1}$$

From equation 1, we derive the following expression for v:

$$v = \frac{2 \times M \times N + N \times K \times h}{mem\_size \times h - 2 \times M \times K} \tag{2}$$

The valid values for h are $\{2, 3, \cdots, M\}$. Algorithm 2 initializes h to 2, and v is then calculated using the formula 2 (Lines 3). Then the number of horizontal slices is increased (h_{temp}) until the best decomposition is achieved, which minimizes the number of slices in matrix B (Lines 5-14). Finally, rows of matrix A are distributed amongst h slices, and columns of matrix B are distributed amongst v slices (Lines 18-21).

4 Experimental Results

In this section, we demonstrate the performance of our out-of-core implementations of matrix multiplication for large dense matrices on GPUs, PHIs and FPGAs. For this purpose, three packages ZZGemmOOC [26], XeonPhiOOC [27] and FPGAOOC [28] have been developed, which use the interfaces defined in *HCLOOC*. We also study the speedup of ZZGemmOOC over NVidia's out-of-core BLAS package CUBLAS-XT [4].

4.1 Evaluation Platform

Our experiments are executed on a server containing an Intel Haswell multicore CPU, NVidia K40c GPU, Intel Xeon Phi, and a Xilinx FPGA (specifications in Tables 1, 2, 3 and 4 respectively). NVidia K40c GPU is provided with three engines including dual copy engines and one kernel invocation engine.

Algorithm 2 Partitioning of matrices A, B, and C using the *Partitioner*

function PARTITIONER(M, N, K, mem_size)
  $h \leftarrow 2$
  $v \leftarrow \lceil \frac{2 \times M \times N + N \times K \times h}{mem\_size \times h - 2 \times M \times K} \rceil$
  $h_{temp} \leftarrow 3$, $v_{temp} \leftarrow \infty$
  while $h_{temp} \leq M$ AND ($v_{temp} \leq 0$ OR $v_{temp} > 1$) **do**
    $v_{temp} \leftarrow \lceil \frac{2 \times M \times N + N \times K \times h_{temp}}{mem\_size \times h_{temp} - 2 \times M \times K} \rceil$
    if $v_{temp} > 0$ **then**
      if $v \leq 0$ OR ($v_{temp} < v$ AND $h \times v \geq h_{temp} \times v_{temp}$) **then**
        $h \leftarrow h_{temp}$
        $v \leftarrow v_{temp}$
      end if
    end if
    $h_{temp} \leftarrow h_{temp} + 1$
  end while
  if $v \leq 0$ OR $v > N$ **then**
    There is no distribution to fit into the accelerator memory, exit.
  end if
  $heights[i] \leftarrow \frac{M}{h}$, $i \in [0, h-1]$
  $widths[i] \leftarrow \frac{N}{v}$, $i \in [0, v-1]$
  $heights[i]++$, $i \in [0, M\%h)$
  $widths[i]++$, $i \in [0, N\%v)$
end function

4.2 Performance of Out-of-core Implementations

We have developed three packages based on *HCLOOC* library which perform out-of-core matrix multiplication of large dense matrices on GPUs, Xeon Phis and FPGAs. For GPU, ZZGemmOOC out-of-core package [26] is developed that reuses CUBLAS for in-core DGEMM invocations. For Xeon Phi, XeonPhiOOC out-of-core package [27] is developed that reuses MKL BLAS [29] for in-core DGEMM invocations. For FPGA, FPGAOOC out-of-core package [28] is designed that reuses a user-defined kernel for in-core invocations. The user-defined kernel calculates matrix multiplication using the straightforward algorithm with three nested loops. The kernel is not fully optimized for FPGA and just uses work item pipelining. All packages use the interface defined by *HCLOOC*. However, they are different in terms of implementation details. For instance, while *Stream Engine* in ZZGemmOOC package is implemented using CUDA, XeonPhiOOC uses Intel offload streams, and FPGAOOC adapts OpenCL command queues. The Intel MKL and CUDA versions used are respectively 2017.0.2 and 7.5.

To evaluate the efficiency of our out-of-core implementations, we measure the execution speed of our packages. The speed of multiplication of two matrices with sizes $M \times K$, $K \times N$ is calculated as $\frac{2 \times M \times K \times N}{t}$ where t represents the execution time, which includes the time taken for matrix multiplication and data transfers from host to device and vice versa.

We show the speeds for ZZGemmOOC and XeonPhiOOC for problem sizes in the set, $\{64^2, 128^2, ..., 44800^2\}$. Since FPGAOOC is very slow for all problem sizes, we evaluate this package for very small problem sizes. To study the

out-of-core computation on the FPGA, the memory size of FPGA is manually set to 64 KB. Matrix sizes used for this implementation is the set, $\{16^2, 32^2, ..., 512^2\}$. In these experiments, when workload size fits into the accelerator memory, all three matrices are transferred to the device, and result is calculated using in-core computations.

Figure 3 compares the speed functions of ZZGEMOOC with CUBLAS-XT on TESLA K40c GPU. In this figure, x-axis represents the size of square matrices and y-axis represents speed in GFLOPS. It is apparent that ZZGemmOOC outperforms CUBLAS-XT for all work-sizes. The gap between ZZGemmOOC and CUBLAS-XT becomes wider as the matrix size grows. The peak double-precision floating point performance of TESLA K40c is 1.43 TFLOPS. The maximum double-precision floating point performance of ZZGemmOOC is 1.17 TFLOPS, which constitutes 82 percent of the peak. ZZGemmOOC provides 1.5x speedup over CUBLAS-XT for small matrix sizes and achieves up to 2.7x speedup for larger ones. The vertical green line in the figure separates the results for in-core matrix multiplication and out-of-core matrix multiplication.

Figure 4 shows the speed function for XeonPhiOOC on Intel Xeon Phi 3120P. We could not find any good third-party implementation for performance comparison. The green line in the figure separates the results for in-core matrix multiplication and out-of-core matrix multiplication. The peak double-precision floating point performance of Intel Xeon Phi 3120P is 1003 GFLOPS. Using XeonPhiOOC, the maximum double-precision floating point performance is 725 GFLOPS, which constitutes 72 percent of the peak.

Figure 5 illustrates the speed function for matrix multiplication on Xilinx Virtex 7 690T FPGA. The green line in the figure highlights the data point where out-of-core computation starts. It is clear that FPGAOOC exhibits no drop in speed for out-of-core computations in comparison with in-core results. There exists no third-party implementation that could be used for performance comparison.

The software implementations of *HCLOOC* presented in this paper can be downloaded from [26], [27] and [28] for GPUs, Xeon Phis and FPGAs, respectively.

5 Conclusion

Cloud service providers are furiously strengthening the pace of their efforts to entice HPC users to migrate their workloads by catering to their unique needs. One such need is the provision of the capability of high performance heterogeneous computing through the provision of hybrid nodes that contain multicore CPUs hosting one or more widely used hardware accelerators such as GPUs, PHIs, and FPGAs.

Hardware acceleration of scientific kernels in cloud poses two prominent challenges. First, virtualization of hardware accelerators must be made possible to meet the core principles of the cloud model, which includes elasticity and on-demand computing. Second, efficient utilization of accelerators to solve

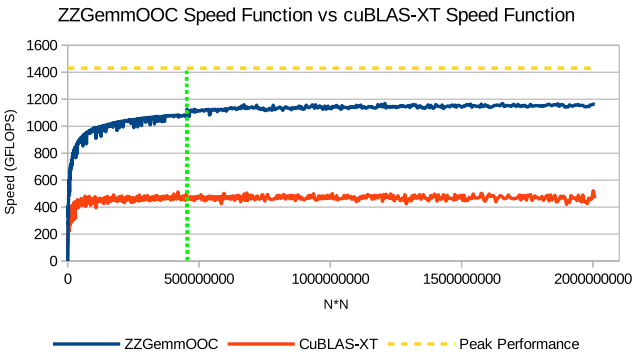

Fig. 3 Comparison of vendor-optimized library CUBLAS-XT with ZZGemmOOC on NVidia K40c GPU. The green line separates in-core computations from out-of-core ones. The dotted yellow line represents the theoretical peak double precision performance of the GPU.

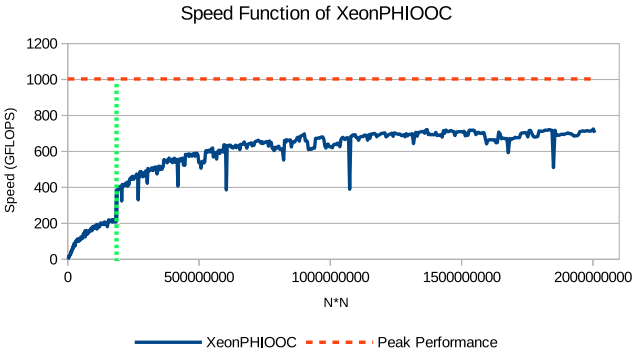

Fig. 4 Speed function of XeonPhiOOC on Intel Xeon Phi 3120P. The green line separates in-core computations from out-of-core ones. The dotted red line represents the theoretical peak double-precision performance.

big instances of data-parallel applications must be facilitated. While virtualization of hardware accelerators at zero performance loss remains a formidable research challenge, we address the second challenge in this paper.

In this paper, we proposed a library containing interfaces (*HCLOOC*) to cope with the limitations that beset execution of data parallel applications for large problem sizes on accelerators including limited main memory size of the accelerator and limited bandwidth of the PCI-E communication link between a host CPU and an accelerator. An optimal software pipelines is adopted for communication-computation overlapping. It is designed using the fundamental building blocks, which are OpenCL command queues for FPGAs, Intel offload streams for Intel Xeon Phis, and CUDA streams that allow concurrent utilization of the copy and execution engines provided in NVidia GPUs.

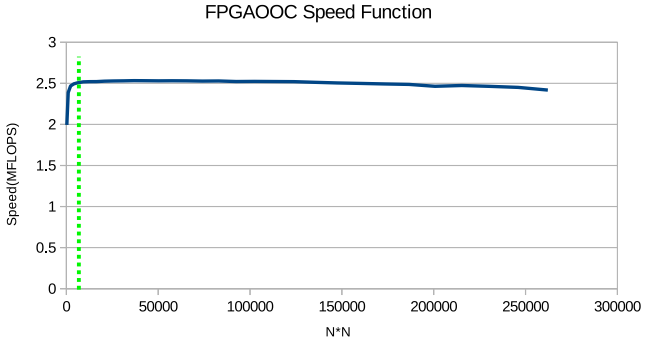

Fig. 5 Speed function of FPGAOOC on Xilinx Virtex 7 690T FPGA. The green line separates in-core computations from out-of-core ones.

The library is evaluated using an out-of-core implementation of matrix multiplication of large dense matrices on a hybrid node, an Intel Haswell multicore CPU server integrated with NVidia K40c GPU, Intel Xeon Phi 3120P, and a Xilinx FPGA. We achieves 82% of peak double-precision floating performance of the GPU and a speedup of 2.7 times compared to NVidia's out-of-core matrix multiplication implementation (CUBLAS-XT). Our experiments showed that the proposed library exhibits 0% drop in performance when the problem size exceeds the main memories of accelerators for GPU, Intel Xeon Phi and Xilinx FPGA.

6 Acknowledgement

This publication has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

References

1. C. K. Filelis-Papadopoulos, E. N. G. Grylonakis, P. E. Kyziropoulos, G. A. Gravvanis, and J. P. Morrison, "Characterization of hardware in self-managing self-organizing cloud environment," in *Proceedings of the 20th Pan-Hellenic Conference on Informatics*, ser. PCI '16. ACM, 2016, pp. 56:1–56:6.
2. T. Lynn, H. Xiong, D. Dong, B. Momani, G. Gravvanis, C. Filelis-Papadopoulos, A. Elster, M. M. Z. M. Khan, D. Tzovaras, K. Giannoutakis, D. Petcu, M. Neagul, I. Dragon, P. Kuppudayar, S. Natarajan, M. McGrath, G. Gaydadjiev, T. Becker, A. Gourinovitch, D. Kenny, and J. Morrison, "CLOUDLIGHTNING: A framework for a self-organising and self-managing heterogeneous cloud," in *Proceedings of the 6th International Conference on Cloud Computing and Services Science - Volume 1 and 2*, ser. CLOSER 2016. SCITEPRESS - Science and Technology Publications, Lda, 2016, pp. 333–338.
3. C. H. Hong, I. Spence, and D. Nikolopoulos, "Fairgv: Fair and fast gpu virtualization," *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2017.
4. CUBLAS-XT, "CUBLAS-XT: Multi-GPU version of CUBLAS library supporting out-of-core routines," 2016. [Online]. Available: https://developer.nvidia.com/cublas

5. S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.
6. Khronos OpenCL Registry. (2017) OpenCL Command Queues. [Online]. Available: https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf
7. Intel. (2017) Programming for Intel MIC architecture. [Online]. Available: https://software.intel.com/en-us/node/684368
8. NVIDIA. (2016) CUDA C Programming Guide. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide/
9. ——. (2013) Tesla K40 GPU accelerator. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf
10. S. Ostermann, A. Iosup, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "A performance analysis of EC2 cloud computing services for scientific computing," in *International Conference on Cloud Computing*. Springer, 2009, pp. 115–131.
11. A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema, "Performance analysis of cloud computing services for Many-Tasks scientific computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 6, June 2011.
12. A. Gupta, L. V. Kal, D. Milojicic, P. Faraboschi, and S. M. Balle, "HPC-Aware VM placement in infrastructure clouds," in *2013 IEEE International Conference on Cloud Engineering (IC2E)*, March 2013, pp. 11–20.
13. M. Parashar, M. AbdelBaky, I. Rodero, and A. Devarakonda, "Cloud paradigms and practices for computational and data-enabled science and engineering," *Computing in Science Engineering*, vol. 15, no. 4, pp. 10–18, July 2013.
14. V. Mauch, M. Kunze, and M. Hillenbrand, "High performance cloud computing," *Future Generation Computer Systems*, vol. 29, no. 6, pp. 1408–1416, 2013.
15. G. Giunta, R. Montella, G. Agrillo, and G. Coviello, *A GPGPU Transparent Virtualization Component for High Performance Computing Clouds*. Springer Berlin Heidelberg, 2010.
16. S. Byma, J. G. Steffan, H. Bannazadeh, A. L. Garcia, and P. Chow, "FPGAs in the cloud: Booting virtualized hardware accelerators with OpenStack," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, May 2014, pp. 109–116.
17. C.-H. Hong, I. Spence, and D. S. Nikolopoulos, "Gpu virtualization and scheduling methods: A comprehensive survey," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 35, 2017.
18. L. Gu, J. Siegel, and X. Li, "Using GPUs to compute large out-of-card FFTs," in *Proceedings of the International Conference on Supercomputing*, ser. ICS '11. ACM, 2011, pp. 255–264.
19. X. Mu, H.-X. Zhou, K. Chen, and W. Hong, "Higher order method of moments with a parallel out-of-core lu solver on gpu/cpu platform," *IEEE Transactions on Antennas and Propagation*, vol. 62, no. 11, pp. 5634–5646, 2014.
20. Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and Multi-GPU systems using functional performance models of Data-Parallel applications," in *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*, 24-28 September 2012, pp. 191–199.
21. Z. Zhong, "Optimization of Data-Parallel scientific applications on highly heterogeneous modern HPC platforms," Ph.D. dissertation, University College Dublin, 2014.
22. J. Wu and J. Jaja, "Achieving native GPU performance for out-of-card large dense matrix multiplication," *Parallel Processing Letters*, vol. 26, no. 02, p. 1650007, 2016.
23. R. Edgar, "SciGPU-GEMM," 2009. [Online]. Available: https://github.com/YaohuiZeng/scigpugemm
24. D. Martin, "High performance computing linpack benchmark for CUDA," 2010. [Online]. Available: https://github.com/avidday/hpl-cuda
25. NVIDIA, "CUDA toolkit documentation," 2017. [Online]. Available: http://docs.nvidia.com/cuda/cublas/index.html#axzz4kRVc2o6B
26. H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky., "ZZGemmOOC: Multi-GPU out-of-core routines for dense matrix multiplization," 2017. [Online]. Available: https://git.ucd.ie/hcl/zzgemmooc.git

27. H. Khaleghzadeh, R. Reddy, Z. Zhong, and A. Lastovetsky., "XeonPhiOOC: Out-of-core package for out-of-core DGEMM on Xeon Phi," 2017. [Online]. Available: https://git.ucd.ie/manumachu/xeonphiooc.git
28. H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky., "FPGAOOC: Out-of-core package for out-of-core dgemm on FPGA," 2017. [Online]. Available: https://git.ucd.ie/hcl/fpgagemm.git
29. Intel MKL BLAS. [Online]. Available: https://software.intel.com/en-us/mkl