IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 14, 2017, accepted May 4, 2017, date of publication May 29, 2017, date of current version June 27, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2708738

A Comparative Study of Programming Environments Exploiting Heterogeneous Systems

BONGSUK KO[1], SEUNGHUN HAN[1], YONGJUN PARK[2], MOONGU JEON[1], AND BYEONGCHEOL LEE[1], (Member, IEEE)

[1]School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Gwangju 61005, South Korea
[2]Department of Computer Science, Hanyang University, Seoul 04763, South Korea

Corresponding author: Byeongcheol Lee (byeong@gist.ac.kr)

This work was supported in part by the National Research Foundation of Korea Grant through the Korean Government (MSIP) under Grant 2015R1C1A1A01052876 and in part by the Institute for Information and Communications Technology Promotion Grant through the Korean Government (MSIP) under Grants R0190-16-2012 and 2017-0-00142.

ABSTRACT This paper compares programming environments that exploit heterogeneous systems to process a large amount of data efficiently. Our motivation is to investigate the feasibility of the adaptive, transparent migration of intensive computation for a large amount of data across heterogeneous programming languages and processors for high performance and programmability. We compare a variety of programming environments composed of programming languages, such as Java and C, memory space models, such as distinct and shared memory, and parallel processors, such as general-purpose CPUs and graphics processing units (GPUs) to examine their performance-programmability tradeoffs. In addition, we introduce a software-based shared virtual memory that creates a view of the host memory inside GPU kernels to enable seamless computation offloading from the host to the device. This paper reveals a programmability-performance hierarchy in which programs increase their performance at the cost of decreasing programmability. The experimental results suggest the desirability of a well-balanced system.

INDEX TERMS Big data processing, heterogeneous systems, programming environment.

I. INTRODUCTION

Big data processing programmers are increasingly confronted by the software engineering challenge of writing and managing well performing programs that process a large amount of data using multiple programming languages, frameworks, and underlying parallel processors [1], [2]. Specifically, such programmers spend a considerable amount of time for choosing and integrating a combination of parallel processors, programming languages, and libraries using foreign function interfaces such as Java Native Interface (JNI) and Python/C [3], [4] and application programming interfaces (APIs) for heterogeneous platforms such as Compute Unified Device Architecture (CUDA) [5] and Open Computing Language (OpenCL) [6]. This selection and integration issue is becoming increasingly non-trivial due to the discrepancy in diverse runtime environments for prevalent big data processing frameworks such as Hadoop [7] and Spark [8] running on managed runtime systems (e.g., JVMs) and emerging high performance accelerator processors such as graphics processing unit (GPU), field-programmable gate array (FPGA), and application-specific integrated circuit (ASIC) optimized for native runtime environments [1]. For instance, to offload computations in Java to these accelerator processors, programmers need to write tedious and error-prone code, which explicitly moves Java objects from the Java heap to the C heap in the device memory through the C heap in the host memory using JNI and CUDA subroutines. In addition, the program risks low runtime performance due to the overhead of crossing both language and processor boundaries. In short, it is challenging to balance the programming productivity and performance.

In order to discover these inefficiencies in more detail, this article studies the performance and programmability tradeoffs in exploiting heterogeneous systems for big data processing. We compare various programming environments composed of programming languages such as Java and C, processors such as CPUs and GPUs, and memory space models such as distinct and shared memory. We first focus on the efficient processing of large amounts of data on the order of tens of giga bytes that can be stored in the host memory of a single server machine but cannot be accommodated in the local memory of a modest GPU. We next introduce and evaluate

a software-based shared memory between the host and device that is a key building block in distributing computation transparently across programming languages and processors.

Our analysis suggests a programmability-performance hierarchy in programming environments in which programs increase performance at the cost of decreasing programmability. For instance, our experimental result shows that the actual computation of dense matrix multiplication achieves the best performance in a complex CUDA program that partitions the input matrices and overlaps the data transfer with the GPU kernel execution using the asynchronous CUDA stream API [5]. This optimization makes the program difficult to maintain, although the computation could be expressed in a few lines of Java code. Worse, when the matrices reside in a Java heap memory, the program must transfer the large matrices from the Java heap to the native heap in the GPU memory through the native heap in the host memory using JNI and CUDA API. Our software-based shared memory exhibits the potential to balance programmability and performance in that it can eliminate the programmability overhead of moving large data sets manually with only a modest performance degradation. We believe that integrating our software-based shared memory with Java GPU JIT compilation [9] would improve both programmability and performance.

II. BACKGROUND

This section overviews various big data frameworks, heterogeneous parallel processors, and programming interfaces, including both potential advantages and potential challenges for their efficient use.

A. BIG DATA FRAMEWORKS

Big data frameworks are extensible and reusable software systems that can both store and process very large data sets. They provide a variety of programming models including map-reduce batch processing in Hadoop, stream processing in Storm and Samza, and hybrid processing in Spark and Flink. These frameworks are written in multiple high-level programming languages such as Java, Scala and Clojure, taking advantage of the automatic memory management and concurrency constructs in Java virtual machines. They naturally take advantage of the parallel execution of instructions in conventional symmetric multi-core processors by distributing computations into dozens and hundreds of work threads accessing the shared virtual memory space.

B. HETEROGENEOUS PARALLEL PROCESSORS

Heterogeneous parallel processors such as GPUs are attached to conventional general purpose processors to accelerate data parallel computations using more than thousands of parallel hardware threads which access high-bandwidth device memory. The discrete GPU structure and high bandwidth requirement separate the device memory from the host memory and introduce two memory space models. A *distinct memory space model* asks programmers to allocate and copy data manually and explicitly between the host and the device using API subroutines (e.g., `cudaMalloc`). A *shared memory space model* such as Unified Virtual Addressing (UVA) [10] creates a view of shared memory between the host and the device, and the hardware and runtime systems automatically either redirect memory accesses or migrate data. For instance, CUDA page locked memory redirects memory access from the GPU to the CPU through direct memory access over a PCIe interconnect, whereas the CUDA managed memory migrates the data between the host and the device in the CUDA managed heap [10].

C. PROGRAMMING INTERFACES

Many programming languages support multi-core and many-core processors with modest extensions. OpenMP extends C, C++, and Fortran with a few directives and clauses to parallelize loop iterations using symmetric multiprocessors. CUDA and OpenCL extend C, C++, and Fortran to offload data-parallel computations to GPUs. These extensions do not directly cover high-level programming languages such as Java, Scala and Clojure, which are widely used in big data frameworks. To take advantage of high-throughput GPUs, programmers need to write boilerplate code for copying data from the Java heap to the C heap in the host memory, and to the C heap in the device memory through JNI and CUDA subroutines. This incurs low programming productivity, and the overhead of moving data across language and hardware boundaries could be also substantial.

III. COMPARING PROGRAMMING ENVIRONMENTS

This section compares several programming environments that exploit heterogeneous systems for big data processing. Each programming environment is composed of processor types (including CPUs and GPUs), programming languages (including Java and C with parallelization extensions), and memory space models (including distinct and shared memory). Programming environments exhibit various degrees of programmability, performance, and device memory limits. Programmability expresses programming effort, which is mainly influenced by the selection of programming languages and memory space models, whereas performance compares computation's execution times. The device memory limit expresses whether a programming environment allows programs to process a large data set in the host memory, which in practice is significantly larger than the device memory.

Table 1 presents and compares programming environments in order of increasing performance at the cost of decreasing programmability and being limited by the capacity of the device memory. General-purpose CPUs and high-level programming language constructs in Java offer high programmability, whereas special purpose GPUs and low-level constructs in CUDA C/C++ allow the programmers to exploit many performance optimization opportunities.

We characterize each programming environment using the dense matrix multiplication computation. Dense matrix

TABLE 1. Performance-programmability tradeoffs in programming environments exploiting heterogeneous systems.

Programming environment			Programmability	Performance	Device memory limit
Processor	Programming language	Memory space model			
CPU	Java and Stream API	Shared memory	high	low	unlimited
CPU	C/C++ and OpenMP	Shared memory	modest	low	unlimited
GPU	C/C++ and CUDA	Page locked shared memory (`cudaMallocHost`)	modest	low	unlimited
GPU	C/C++ and CUDA	Managed shared memory (`cudaMallocManaged`)	modest	modest	limited
GPU	C/C++ and CUDA	Distinct memory	low	high	limited
GPU	C/C++ and CUDA	Distinct memory with CUDA Stream API	low or infeasible	high	modest

multiplication takes two input matrices A and B with N rows and columns and produces an output matrix C of N rows and columns such that each entry in C at ith row and jth column is the inner product of the ith row in A and jth column in B. Figure 1 presents code snippets representing the dense matrix multiplication in the six programming environments. The two representations in Figures 1(a) and 1(b) utilize the host parallel CPU processors with a few lines of very concise statements in part from using parallel language constructs such as Java 8 Stream API and lambda expressions and OpenMP directives. The two representations in Figures 1(c) and 1(d) utilize GPUs, separate the host and device codes, and illustrate asymmetric shared memory models in which both the host and kernel codes access the three matrices without any explicit data move operations. `cudaMallocHost` in Figure 1(c) allocates a page-locked memory that is resident in the host memory so that all memory accesses from the GPU kernel are redirected to the host memory through the PCIe channel, whereas `cudaMallocManaged` allocates a managed memory that is resident in the device memory in practice. The last two representations in Figures 1(e) and 1(f) exhibit source lines for allocating device memory and transferring matrices between the host and device memories when the shared memory model is not used. They are distinguished by the degree of concurrency exploited. The pure distinct memory asks the programmers to transfer matrix objects between the CPU memory and GPU memory using `cudaMemcpy`, whereas the use of CUDA Stream API offers fine-grained control allowing the overlap between the data transfer and kernel execution.

The six parallel programs exhibit a programmability-performance hierarchy in which programmers increases performance at the cost of decreasing programmability. Low-level programming languages such as C/C++ and APIs such as CUDA Stream decrease the programmability. For instance, the matrix multiplication program increases the number of source lines containing low-level constructs such as a separate GPU kernel code, scheduling parameters (e.g., `schedule(dynamic)` in OpenMP), the numbers of grids and threads, and concurrent memory transfer, all of which require careful examination for correct use. However, these constructors exploit low-level hardware constructors (e.g., PCIe bandwidth and the massive number of cores in GPUs) as much as possible and increase the computation throughput. For instance, the programming environment illustrated in Figure 1(f) exploits parallelism and concurrency in computation and communication, but it is non-trivial or infeasible to employ the CUDA Stream API for arbitrary programs.

Device memory limit is another critical issue in processing a large amount of data in a balanced environment in which the host memory is significantly larger than the device memory. All programming environments with only CPUs are free from any device memory limit because they process the data in the host memory. Programming environments with GPUs show different degrees of device memory limit. Page locked shared memory does not have a device memory limit because the device code directly accesses the data objects resident in the host memory. Managed shared memory exhibits mixed results, where legacy GPU devices generate an out-of-memory error upon allocating a large data object that exceeds the device memory size, whereas the recent high-end Tesla P100 enables such memory oversubscription [11]. The CUDA Stream API relaxes this device memory limit by asking the programmers to partition the large input data into a set of small data and copy these pieces of data asynchronously with the kernel execution.

For processing arbitrary big data, the first three environments in Table 1 are feasible at the cost of under-utilizing the high-throughput GPUs. The first two environments just exclude GPUs, and the third one shows prohibitively low performance because all data accesses in the kernel code generate data requests in the host memory through low-bandwidth PCIe channels. This inadequacy motivates us to design a shared virtual memory between CPUs and GPUs [12]–[15].

IV. SOFTWARE-BASED SHARED VIRTUAL MEMORY

This section introduces a software-based shared virtual memory (SBSVM) to create a view of the host memory inside GPU kernels using CPU-GPU remote procedure calls.

A. ASYMMETRIC SHARED MEMORY SPACE

An asymmetric shared memory space allows GPU kernels to access arbitrary objects in the host memory through a set of indirect references to the objects in the host memory. The programming interface is composed of a C/C++ opaque data type of `Cptr` and a set of access functions. A `Cptr` value in the device code is an indirect reference to an object in

```
IntStream.range(0, N * N).parallel().forEach(
    idx->{
  int i=idx/N, j=idx % N;
  for(int k=0;k<N;k++)
    sum += B[i*N+k] * C[k*N+j];
  A[idx]=sum;
```

(a)

```
  #pragma omp parallel for schedule()
  for(i=0;i<N:i++)
    for(j=0;j<N;j++)
      for(k=0;k<N;k++)
        c[i][j]+=a[i][k]*b[k][j];
```

(b)

```
//host
  ...
  //Malloc part
  cudaMallocHost((void**)&A, size);
  cudaMallocHost((void**)&B, size);
  cudaMallocHost((void**)&C, size);
  ...
  //Initialize Matrix
  ...
  dim3 Grid(512,1,1);
  dim3 threads(512,1,1);
  kernel <<<Grid, threads>>>(A,B,C);

//device (kernel)
  int row = blockIdx.y*blockDim.y
          +threadIdx.y;
  int col = blockIdx.x*blockDim.x
          +threadIdx.x;
  if(row>N||col>N) return;
  for(int i=0;i<N;i++)
    sum+= A[N*row+i]*B[i*N+col];
  C[row*N+col]=sum;
```

(c)

```
//host
  ...
  //Malloc part
  cudaMallocManaged((void**)&A, size);
  cudaMallocManaged((void**)&B, size);
  cudaMallocManaged((void**)&C, size);
  ...
  //Initialize Matrix
  ...
  dim3 Grid(512,1,1);
  dim3 threads(512,1,1);
  kernel <<<Grid, threads>>>(A,B,C);

//device (kernel)
  int row = blockIdx.y*blockDim.y
          +threadIdx.y;
  int col = blockIdx.x*blockDim.x
          +threadIdx.x;
  if(row>N||col>N) return;
  for(int i=0;i<N;i++)
    sum+= A[N*row+i]*B[i*N+col];
  C[row*N+col]=sum;
```

(d)

```
//host
  ...
  //CPU Malloc part
  float hA = (float *)malloc(size);
  float hB = (float *)malloc(size);
  float hC = (float *)malloc(size);
  ...
  //Initialize Matrix
  ...
  //Cuda Malloc part
  float *dA, *dB, *dC;
  cudaMalloc(&dA, size);
  cudaMalloc(&dB, size);
  cudaMalloc(&dC, size);
  //Memcpy part (H2D)
  cudaMemcpy(dA,hA, size ,H2D);
  cudaMemcpy(dB,dB, size ,H2D);
  //Kernel Execution
  dim3 Grid(512,1,1);
  dim3 threads(512,1,1);
  kernel <<<Grid, threads>>>(dA,dB,dC);
  ...
  //Memcpy part(D2H)
  cudaMemcpy(hC,dC, size ,D2H);
  ...
//same device function to Code 3.
```

(e)

```
//host
  ...
  //Allocation pinned memory
  cudaMallocHost ((void**)&A, size);
  cudaMallocHost ((void**)&B, size);
  cudaMallocHost ((void**)&C, size);
  ...
  //Initialize Matrix
  ...
  //Cuda Malloc part
  float *dA, *dB, *dC;
  cudaMalloc(&dA, size);
  cudaMalloc(&dB, size);
  cudaMalloc(&dC, size);
  //Memcpy part (H2D)
  for(i=0;i<2;i++)
  {
    cudaMemcpyAsynch(dA,A, size ,H2D, stream[i]);
    cudaMemcpyAsynch(dB,B, size ,H2D, stream[i]);
  }
  //Kernel Execution
  dim3 Grid(512,1,1);
  dim3 threads(512,1,1);
  for(i=0;i<2;i++){
    kernel<<<Grid, threads,0,stream[i]>>>(dA,dB,
        dC);
    //Memcpy part(D2H)
    cudaMemcpyAsynch(C,dC, size ,D2H, stream[i]);
  }
  ...

//same device function to Code 3.
```

(f)

FIGURE 1. Representations of multiplying dense matrices in the six programming environments. (a) CPU, Java and Stream API, and shared memory. (b) CPU, C and OpenMP, and shared memory. (c) GPU, C/C++ and CUDA, and page locked shared memory. (d) GPU, C/C++ and CUDA, and managed shared memory. (e) GPU, C/C++ and CUDA, and distinct memory. (f) GPU, C/C++ and CUDA, and distinct memory with CUDA Stream.

```
int row = blockIdx.y*blockDim.y + threadIdx.y;
int col = blockIdx.x*blockDim.x + threadIdx.x;
if(row>N||col>N) return;
float sum = 0;
CPtr<float> a = &A[row*N];
CPtr<float> b = &B[col];
for(int i=0;i<N;i++) {
  sum+= a.read() * b.read();
  ++a;
  b+=N;
}
Cptr<float> c = &C[row*N+col];
c.write(sum);
```

FIGURE 2. GPU kernel code of accessing the host memory through `Cptr`.

the host memory. These opaque references disallow low-level pointer operations such as pointer dereference and arithmetic. Instead, the device code executes numerous access functions to read and write the values in the host memory. Figure 2 shows a GPU kernel code snippet multiplying two dense matrices resident in the host memory using the `Cptr` references. This kernel code is identical to the kernel routines in Figure 1 except that all expressions of accessing matrix elements are replaced with calls to the `Cptr` access function, and a few `Cptr` variables `a`, `b` and `c` are declared as indirect references to the elements in the input and output matrices.

To access the host memory objects without preempting GPU kernel execution, to reduce data transfer overhead, and to exploit memory reference locality, our shared memory design adapts the page caching, address translation, and the CPU-GPU remote procedure call in RSVM [15] and Active-Pointers [14]. Our shared memory system is composed of CPUPointer, page cache, and CPU-GPU remote procedure call. *CPUPointer* is the implementation of the `Cptr` interface to process all requests from the GPU kernels to access the host memory objects. The *page cache* manages a set of cached host pages and mapping from their host addresses to the device addresses. The *CPU-GPU remote procedure call* exchanges pages between the host memory and the device memory.

A memory access goes through some of these four subsystems. In the fast path case, in which a `CPtr` reference points to an object in the cached page in the page cache, the overhead includes checking the condition bit of validating the cached condition (page hit). When the condition bit is invalid and the object is in the cached page, the slow path successfully looks up the page cache, updates the validity bit, and accesses the cached object in the page cache (minor fault). In the worst case, in which the object is not resident in the page cache, the slowest path requests a CPU-GPU remote procedure call of copying the host page to the page cache, updates the page cache and validity condition bit, and accesses the cached object in the page cache (major fault). When the page cache is in use for the other object, the extra path temporarily allow the read-through and write-through operations (hash collision).

1) CPUPointer

`Cptr` is an unsigned integer type containing two members: a valid bit and the mapping data. The valid bit contains the two states of a `Cptr` reference: the *linked* state and the *unlinked* state. In the linked state, the cached host page resides in the page cache, and the mapping data is the device address of the host object in the page cache. We call this location an internal address, or iAddress. A memory access through a linked CPUPointer locates the cached object (page hit). In the unlinked state, the mapping data expresses the host address of the object, and we call this location an external address or xAddress. An access through an unlinked reference activates a page fault handler to service either a minor fault or a major fault depending on the state of the page cache.

2) PAGE CACHE

Page cache keeps a poll of the cached host memory pages inside the device memory and a hash table of mapping from the host page addresses to the cached states. Each cached state contains a pointer to the cached page in the device memory, a dirty bit for writing the updated host page back to the host memory, and an integer value to count the number of `Cptr` references to the cached page. The page fault handler looks up the hash table with the external host address in an unlinked reference. In the event of a lookup success (minor fault), the page fault handler increments the reference count in the caches state entry, installs the internal address in the state entry into the `Cptr` reference, and turns the reference into the linked state. In the event of a lookup failure (major fault), the page fault handler requests to transfer the page from the host memory to the device memory through the CPU-GPU remote procedure call and handles the minor fault.

3) REMOTE PROCEDURE CALL

The CPU-GPU remote procedure call serves asynchronous data transfer requests from the GPU kernel between the host and device memories. The data transfer granularity is a memory block of 4 KB pages. It is composed of an RPC server running in the host thread and RPC clients running in the GPU kernel threads. Once the host thread activates a GPU kernel, it executes the RPC server, which waits for a request from an RPC client. The major page fault handler running in a GPU thread enqueues data transfer requests. For the communication channel and data transfer, the RPC subsystem allocates and updates the page locked memory using the `cudaMallocHost` and `cudaMemcpyAsync` subroutines in CUDA.

B. MANAGING CONCURRENCY

Our shared memory subsystem manages concurrency within kernel threads with several concurrency idioms. The fast path of page hits is free from any synchronization on the page cache and RPC channels. The slow path of minor and major

faults employs fine-grained locks and lock-free synchronizations for the hash table in the page cache and RPC channels. We carefully address the potential problem of *divergence deadlock* where a group of threads in a *warp* cannot make progress when the lock owner thread cannot release the lock when it becomes idle because of a branch divergence [5]. There are two methods to solve this deadlock problem: serializing the execution of the critical sections and using one lock for each warp and critical section. The former method is used to lock a cached state in the page cache hash table in handling the major fault, whereas the latter method locks an element of the RPC queue in handling the major page fault.

```
bool done = false;
while (!done) {
  if (!page->read_mode) {
    if (lock(page) == LOCKED) {
      if (!page->read_mode) {
        while (page->readers != 0) {
          __threadfence();
        }
        if (is_count_zero(page)) {
          major_handler(page);
          done = true;
        }
        atomicExch(&page->read_mode, true);
      }
      unlock(page);
    }
  } else {
    atomicAdd(&page->readers, 1);
    if (page->read_mode) {
      int result = minor_handler(page);
      if (result == NEED_PAGE_SWAP) {
        if (page->count == 0) {
          atomicExch(&page->read_mode, false);
        } else {
          extra_handler();
          done = true;
        }
      } else {
        done = true;
      }
    }
    atomicSub(&page->readers, 1);
  }
}
```

FIGURE 3. The page fault handler to solve the readers-writers problem.

There is a readers-writers problem on handling the minor and major fault for each page cache; the major faults are the side of writers and the minor faults are the side of readers. And it should be a readers-preference to higher cache availability. The solution that is achieved to avoid divergence deadlock is shown in Figure 3.

The loop of the outside enables the serializing the execution of the critical sections. Even though some threads in a warp fail to obtain the locks, the other threads can continue to execute the critical section, and the failed threads can try again to obtain the locks at next iteration. The condition variable of `read_mode` enables to control the execution flow of entering the readers' section. The `read_mode` is updated by the atomic operation, and the variable is utilized as the double-checked locking mechanism. Thanks to this double-checked locking mechanism, any spin-lock is not required in the readers' section, which the spin-lock may incur the divergence deadlock. The value of the `readers` counts the threads that entered the section of reader, and the value of `readers` is updated by atomic operation. The busy-lock, which uses `reads`, makes a writer to wait until all readers exit the readers' section, and this feature shows that it is a readers-preference. Since the page cache is in the page mode, the value of `readers` is guaranteed to decrease into zero, and therefore, it's free from the problem of dead lock.

V. METHODOLOY

This section describes benchmarks and the measurement setup to evaluate performance of programming environments.

TABLE 2. Summary of the benchmarks.

Benchmark	Summary	Data type
MM	Dense matrix multiplication: $C = AB$	float
Gemm	Dense matrix multiplication from PolyBench [16]: $C = \alpha AB + \beta C$, ported to Java	int
Gesummv	Scalar, vector and matrix multiplication from PolyBench [16], ported to Java	int
Blackscholes	Financial application, which calculates the price of European put and call options	double
Series	Series from the Java Grande Benchmarks [17]	double
SpMM	Sparse matrix multiplication from the Java Grande Benchmarks [17]	double

A. BENCHMARKS

Table 2 presents six benchmarks to compare performance for each programming environment. The *benchmark* column lists the benchmarks' names, the *summary* column summarizes the computations in the benchmarks, and the last column shows the data types in the computations. `MM` is a dense matrix multiplication program taking two square matrices. `Gemm` is a matrix calculation program taking three square matrices and two scalar values and producing an output square matrix. `Gemm` has the identical complexity with `MM`, but it consumes more memory space than `MM`. `Gesummv` performs a matrix computation taking two square matrices, two scalar values, and two vectors and producing a vector. `Gesummv` requires a less amount of computation complexity than `MM` and `Gemm` because it multiplies vectors with matrices. `Blackscholes` is a financial application taking vectors and producing vectors to calculate put options. `Series` performs a vector calculation taking one vector of N elements and producing a vector of $2N$ elements to compute Fourier coefficients. `Series` uses significantly less memory space than `Blackscholes`. `SpMM` is a sparse matrix multiplication program taking one sparse matrix and one vector and producing a vector. `SpMM` requires locking operations to avoid race conditions while others do not need any synchronization.

Figure 4 reports communication ratio for each benchmark. The horizontal axis represents benchmarks, and the vertical bars show the three components of their execution times: (1) kernel execution for computation times in the kernel

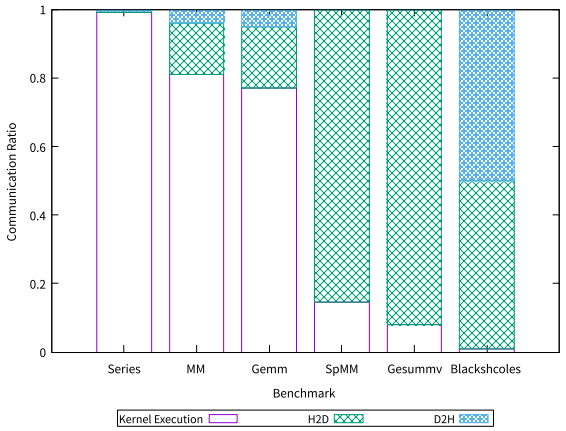


FIGURE 4. Normalized execution times of communication and computation.

subroutines, (2) D2H for transferring data from the device (GPU) to the host (CPU), and (3) H2D for transferring data from the host (CPU) to the device (GPU). `Series`, `MM`, and `Gemm` are computation intensive because the fraction of their kernel execution is more than 0.7. On the other hand, `SpMM`, `Gesummv` and `Blackscholes` spend a large fraction of their execution on transferring data between the host and the device.

B. SETUP

We run all benchmark programs on a host machine with two 2.7GHZ 12-core Intel Xeon E5-2697 v2 processors, 66GB main memory, and a 1GHz GTX 980TI GPU device with 2,816 CUDA cores and 6GB memory. The machine runs Linux kernel 3.13.0 in the Ubuntu 16.04 distribution operating system. We implement each benchmark under the programming environments in Table 1 and vary the input size for each benchmark to cover memory sizes from a few mega bytes to dozens of giga bytes. Each benchmark reports wall clock times in milliseconds from the start to the completion of the computation. To tolerate variations in nondeterministic timing runs, we report the average of the last five executions out of 12 iterations. When a benchmark runs for more than 8 hours, it reports a timeout.

VI. EVALUATION

This section evaluates and analyzes the performance of the programming environments.

A. THROUGHPUT

Figure 5 reports the throughputs of the benchmark runs for the various programming environments and input sizes. The horizontal axis shows the input data size, and the vertical axis shows the execution times of the benchmarks under various programming environments. We apply a log scale to both axes to acount for the large variations in the input size and execution time. Owing to either timeouts or out-of-memory errors, some programming environments may not show data points for large amounts of input data. All data points exhibit the pattern of an increasing the execution time as the input size increases.

`MM` and `Gemm` show similar performance results. The two programming environments employing CPUs perform worse than the those employing GPUs or report timeouts. This suggests that GPUs are desirable for intensive computations with large data sets. For small data sets from 1K to 4K, the conventional CUDA programming with distinct memory and managed shared memory performs the best, whereas large data sets from 8K to 32K present programming challenges. The programming environment employing CUDA Stream API performs the best, but this programming environment may not express other kinds of computation as illustrated in Section III. SBSVM processes this very large data set at a somewhat decreased rate with respect to CUDA managed memory. Figures 6a and 6b report composition of the execution times in MM and Gemm when the inputs are 1K-by-1K matrices. SBSVM does not trigger any significant communication activities, but its kernel execution time is slower than the other two programming environments. This slowdown comes from frequent minor faults which occur when scanning the column vectors of the second input matrix with little spatial locality. We believe that this slowdown will be eliminated as we mature SBSVM.

`Gesummv` is a communication intensive benchmark consuming 90% of its execution time on data transfer as shown in Figure 4. The programming environments employing CPUs outperform the other environments employing GPUs over a wide range of large data sets. The GPU programming environments with distinct memory and managed memory have similar performance capabilities, but they report out-of-memory errors due to the limit of device memory when the input size is 32K. However, SBSVM processes this 32K input size while it performs as good as the other programming environments employing GPUs. Figure 6c compares the composition of the execution times in three programming environments when the input size is 4K. Distinct memory requires a significant amount of time for data transfer while managed memory needs less time for data transfer. SBSVM spends a smaller amount of time on transferring data than the others, but its kernel execution performs the worst. As a result, SBSVM performs as good as distinct memory and managed shared memory while it overcomes the device memory limit.

`Blackscholes` exhibits distinguishable performance results where SBSVM performs the best over all input data sets. In addition, page locked shared memory outperforms distinct memory and managed shared memory. Figure 6d compares the composition of the execution times in four programming environments. SBSVM and page locked shared memory show slow kernel execution times while managed shared memory and distinct memory show large amounts of overhead for memory communication. The net result is that page locked shared memory and SBSVM perform better than the others. This result suggests the performance potential of

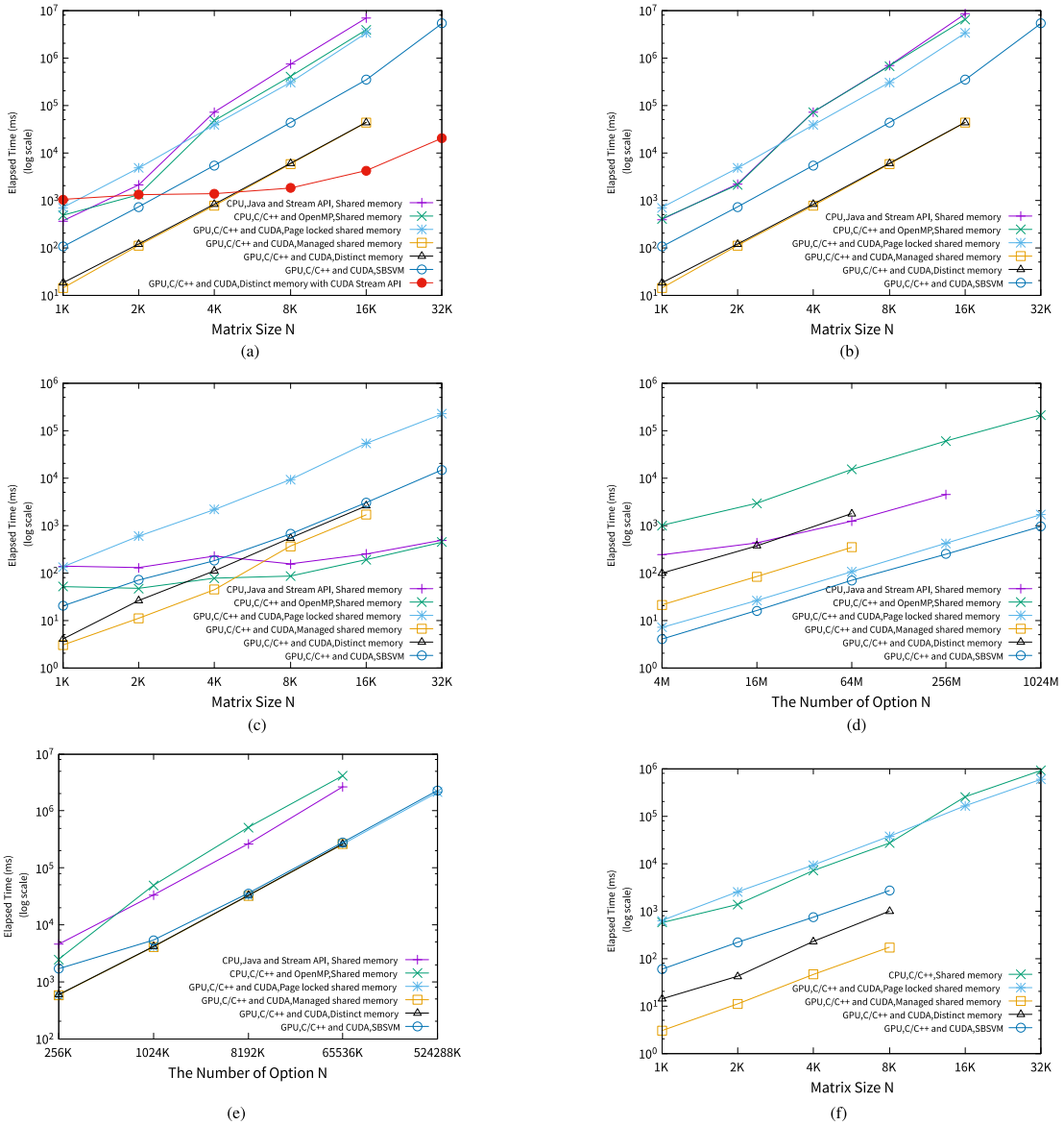


FIGURE 5. Performance of programming environments. (a) MM. The three programming environments labeled by "CPU,Java and Stream API,shared memory," "CPU,C/C++ and OpenMP," and "GPU,C/C++ and CUDA,Page locked shared memory" report timeouts when the input size *N* is larger than 16K. The two programming environments labeled by "GPU,C/C++ and CUDA,Managed shared memory" and "GPU,C/C++ and CUDA,Distinct memory" report out-of-memory errors when the input size *N* is larger than 16K. (b) Gemm. The three programming environments labeled by "CPU,Java and Stream API,shared memory," "CPU,C/C++ and OpenMP," and "GPU,C/C++ and CUDA,Page locked shared memory" report timeouts when the input size *N* is larger than 16K. The two programming environments labeled by "GPU,C/C++ and CUDA,Managed shared memory" and "GPU,C/C++ and CUDA,Distinct memory" report out-of-memory errors when the input size *N* is larger than 16K. (c) Gesummv. All programming environments labeled by "GPU,C/C++ and CUDA,Managed shared memory" and "GPU,C/C++ and CUDA,Distinct memory" report out-of-memory errors when the input size *N* is larger than 16K. (d) Blackscholes. The two programming environments labeled by "GPU,C/C++ and CUDA,Managed shared memory" and "GPU,C/C++ and CUDA,Distinct memory" report out-of-memory errors when the input size *N* larger than 64M. (e) Series. The two programming environments labeled by "CPU,Java and Stream API,shared memory," and "CPU,C/C++ and OpenMP," report timeouts when the input size *N* is larger than 65,536K. The two programming environments labeled by "GPU,C/C++ and CUDA,Managed shared memory" and "GPU,C/C++ and CUDA,Distinct memory"report out-of-memory errors when the input size *N* larger than 65,536K. (f) SpMM. The two programming environments labeled by "GPU,C/C++ and CUDA,Managed shared memory," and "GPU,C/C++ and CUDA,Distinct memory" report out-of-memory errors when the input size *N* larger than 8K.

SBSVM when the benchmark has large amounts of overhead for memory communication.

`Series` is a computation intensive benchmark consuming 99% of it execution time for kernel execution as show in Figure 4. Not surprisingly, the GPU programming environments outperform the other CPU programming environments, which report timeouts when the input data size is 524,288K. SBSVM and page locked shared memory overcome the device memory limit while distinct memory and managed shared memory report out-of-memory errors.

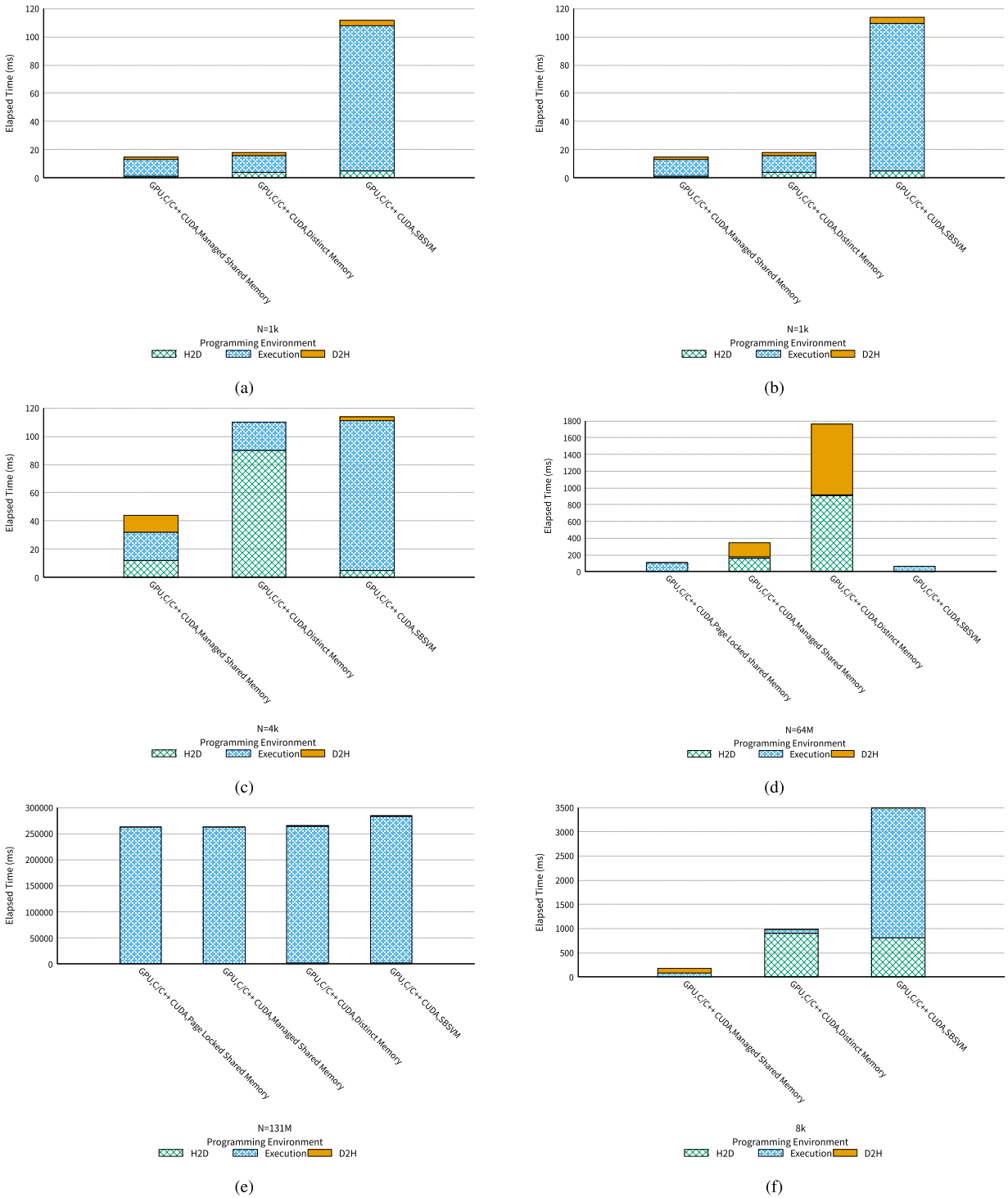


FIGURE 6. Composition of execution times. (a) Elapsed time for MM, Input size is 1K. (b) Elapsed time for Gemm, Input size is 1K. (c) Elapsed time for Gesummv, Input size is 8K. (d) Elapsed time for Blackscholes, Input size is 64M. (e) Elapsed time for Series, Input size is 131M. (f) Elapsed time for SpMM, Input size is 8K.

Figure 6e compares the composition of the execution times in four programming environments. All GPU programming environments show similar performance.

SpMM is a communication intensive benchmark. It is distinguishable from others in that it employs locks to synchronize concurrent accesses to the elements in matrices. GPU programming environments outperform CPU programming environments over all data sets. Managed memory performs the best, and the performance gap between managed memory and SBSVM is quite significant. The random

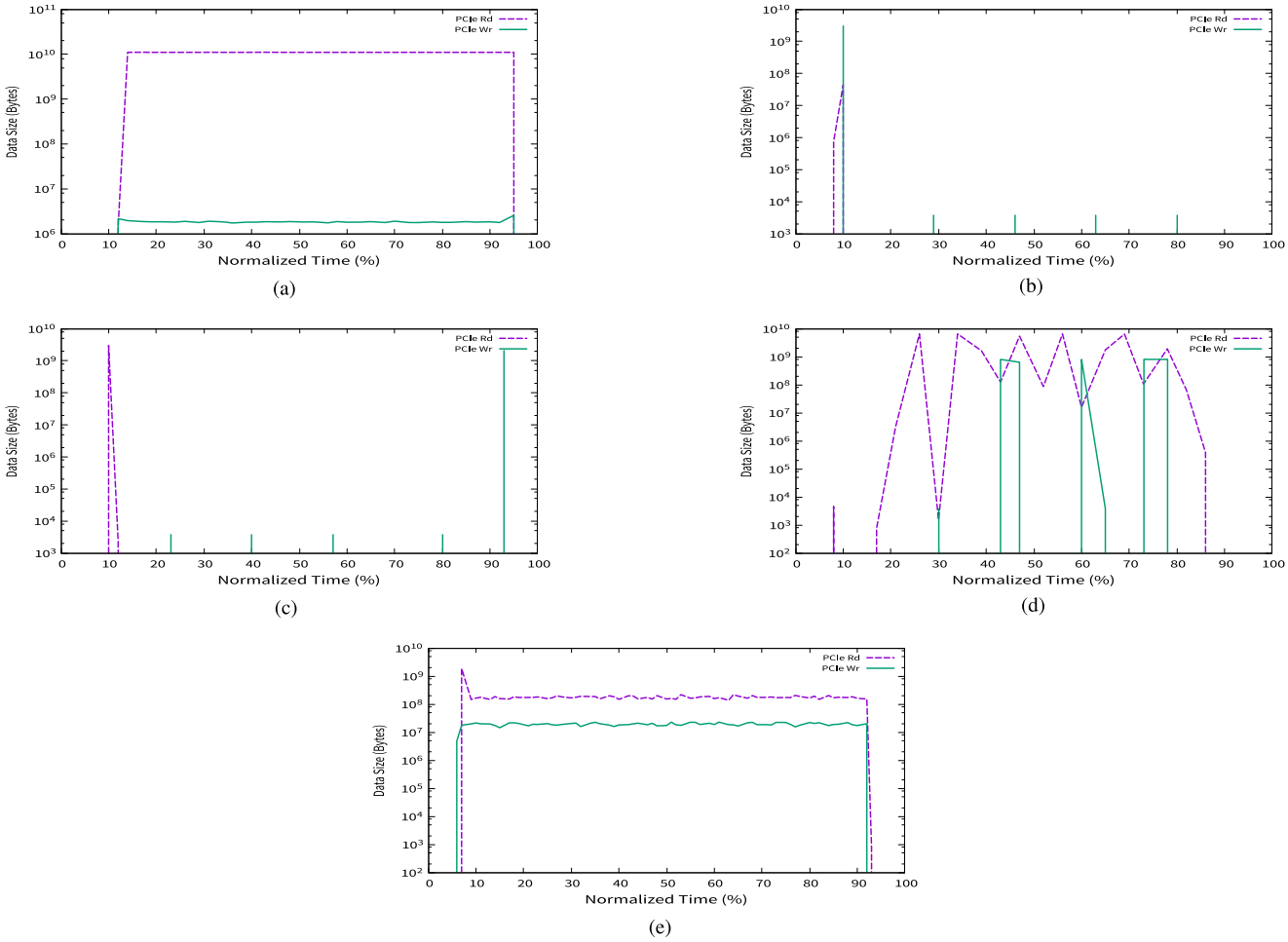


FIGURE 7. Time series of data transfer rates over the PCIe channel for MM. (a) Page locked shared memory. (b) Managed shared memory. (c) Distinct memory. (d) Distinct memory with CUDA stream. (e) PCIe result for CUDA with cache.

memory access pattern seems to show low spatial locality and trigger frequent page faults in SBSVM. This benchmark seems to identify the overhead of the address translation in SBSVM.

Based on these evaluation results, SBSVM is proven to be an effective alternative of current GPU shared memory schemes as it generally shows comparative performance to CUDA distinct memory while maintaining programability as shown in most benchmarks (`MM`, `Gemm`, `Gesummv`, `and SpMM`). This is mainly because SBSVM highly reduces the memory coherence traffic between CPU and GPU memory structures. Moreover, it often shows the best performance for several benchmarks when the target workload requires 1) large memory transfer overhead (`Blackscholes`) or 2) larger data set more than the GPU memory size (`Series`). Therefore, SBSVM can efficiently resolve the distinct memory space problem of the traditional GPU programming model with a minimum performance overhead.

B. HOST AND DEVICE COMMUNICATION

To characterize the communication patterns between the host and device in the programming environments, Figure 7 reports the time series of the data transfer rate in bytes per second for read and write requests over the PCIe channel while executing the benchmarks for `MM`. The horizontal axis normalizes the execution time and the vertical axis shows the data size on a log scale. The page-locked shared memory in Figure 7a transfers a large amount of data continuously because all memory accesses in the kernel code trigger read and write requests over the PCIe channel. The managed shared memory model in Figure 7b triggers a data transfer in the beginning as the data are resident in the device memory and because all memory accesses from the host during the initialization phase trigger PCIe traffic. The distinct memory model in Figure 7c triggers PCIe read requests in the beginning when the program manually moves the input matrices from the host memory to the device memory. There are no data transfer activities while the kernel is running. In the end, this triggers write-back traffic when the host program moves the output matrix from the device to the host. The distinct memory with CUDA streaming in Figure 7d exhibits several peaks in read and write requests because it transfers submatrices between the host and device memory while executing the kernels several times. SBSVM shows a pattern of

continuous read and write requests similar to the pattern in page locked memory because its major fault handler requests data continuously over the RPC channel. The difference is that SBSVM consumes less PCIe bandwidth because it does not request data transfers when data appears in the page of the device memory.

VII. THREATS TO VALIDITY

The goal of our comparative study is to examine the performance and programmability tradeoffs in a variety of programming environments and investigate the feasibility of adaptive, transparent migration of Java computation to heterogeneous processors such as GPUs for big data processing. Our comparative study is based on 36 programs exhibiting the six benchmarks in Table 2 using the seven programming environments including the six environments in Table 1 and SBSVM in Section IV.

The seven programming environments and six benchmarks do not cover all possible programming environments and benchmarks. However, they cover several prevalent programming languages such as Java and C and memory space models such as shared and distinct memory for big data processing frameworks written in Java [7] and heterogeneous processors [5]. The six benchmarks involve matrix and vector data, which can be easily paralleled using data parallel processors such as GPUs while big data computation would process a wide variety of data types as well as matrix and vector. Further evaluation with more environments and benchmarks remains as future work.

The programmability of each programming environment is a subjective metric, which is affected how benchmarks organize their computation and communication and how programmers accept low-level features such as CUDA stream API. Our qualitative analysis assumes that a programmer's productivity gets low when spending extra time on thinking of how to move and deallocate data and writing the source lines of managing data manually.

The performance of each programming environment is a quantitative metric, which is directly measured by comparing execution times of benchmarks over a number of inputs. This comparison assumes that benchmark programs is optimized equally well over all programming environments. This assumption is likely to be valid because the six benchmark programs express simple computations with dozens of source lines in Java and C.

VIII. RELATED WORKS

A. PROGRAMMING ENVIRONMENTS

Heterogeneous systems increasingly cover a wide variety of programming languages including C/C++, Python, and Java. CUDA and OpenCL extend C/C++ with a few keywords and provide a number of API routines to allow programmers to offload data-parallel computations to GPUs. Language bindings such as jCUDA and pyCUDA directly wrap the API routines in Java and Python, and the kernel routines are expressed as string values in Java and Python. The kernel routines in C/C++ bypass any syntax and type checking, and they cannot execute any routine written in Java and Python. Dynamic compilers [9], [18], [19] relax these limitations. To the best of our knowledge, no prior work compares programming environments for big data processing as we do.

B. MEMORY MANAGEMENT

CUDA has naturally evolved its programming model from a distinct memory space to shared memory as it is increasingly used to process a large variety of data. CUDA 4 introduces unified virtual addressing (VUA) to allow both the host and the device memories to be addressed together in a single address space. CUDA 6 takes the next step of introducing a unified memory as a pool of memory that is migrated between the host and device automatically when the memory is allocated through a CUDA runtime function (`cudaMallocManaged`). Several studies have explored implementations of the shared memory space between CPUs and GPUs at multiple levels, including hardware [12], compiler [13], runtime library [14], [15], [20], and languages [21]. These studies inspired our software-based shared virtual memory, which adapts the bimodal references, page cache, and remote procedure calls in ActivePointers [14]. While ActivePointers focuses on accessing large data sets in the storage devices through the `mmap` interface, SBSVM focuses on accessing data objects in the host memory from the device kernel code. RSVM [15], GMAC [20], and CUDA unified memory ask the host code to allocate shared memory explicitly through API functions, including `rsvm_malloc`, `adsmAlloc`, and `cudaMallocManaged`. GMAC [20] and CUDA unified memory avoid the overhead of translating address values between host and device memory by ensuring that shared objects are placed at the identical virtual address in the two distint memories. SBSVM does not have the restriction of allocating shared objects using a custom memory allocator. Instead, it performs address translation in kernel executions with a modest overhead.

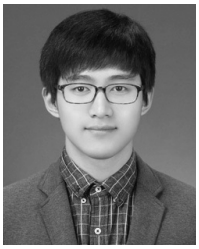
IX. CONCLUSION

This article compares programming environments that exploit heterogeneous systems for processing large data sets. Our programming environments cover two processor types of CPUs and GPUs, multiple programming language interfaces such as Java, C/C++, Stream API, OpenMP, and CUDA, and two memory space models including both distinct and shared memories. In addition, we present a software-based shared virtual memory to create a view of the host memory within the device kernel. We compare these programming environments over a few benchmarks while varying the input size. We validate the performance-programmability hierarchy, in which a high performance for a large data set is achieved at the cost of decreasing programming productivity. To balance performance and programmability, an efficient implementation of the shared memory space would potentially approach the throughput of distinct memory models with stream API.

REFERENCES

[1] D. A. Reed and J. Dongarra, "Exascale computing and big data," *Commun. ACM*, vol. 58, pp. 56–68, Jun. 2015.
[2] A. Avila, A. Maron, R. Reiser, M. Pilla, and A. Yamin, "GPU-aware distributed quantum simulation," in *Proc. ACM Symp. Appl. Comput. (SAC)*, 2014, pp. 860–865.
[3] S. Liang, *The Java Native Interface: Programmer's Guide Specification*. Boston, MA, USA: Addison-Wesley, 1999.
[4] Python Software Foundation. *Python/C API Reference Manual*, accessed on Apr. 2017. [Online]. Available: https://docs.python.org/3/capi
[5] *CUDA C Programming Guide 8.0*, N. Corporation, Santa Clara, CA, USA, 2016.
[6] *The OpenCL Specification Version: 2.1*, document 23, KOW Group, 2015.
[7] *Apache Hadoop*, accessed on Apr. 2017. [Online]. Available: http://hadoop.apache.org
[8] *Apache Spark*, accessed on Apr. 2017. [Online]. Available: http://spark.apache.org
[9] K. Ishizaki, A. Hayashi, G. Koblents, and V. Sarkar, "Compiling and optimizing java 8 programs for GPU execution," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, 2015, pp. 419–431.
[10] *Cuda C Programming Guide 8.0*, N. Corporation, 2016.
[11] *Nvidia Tesla P100: Infinte Compute Power for the Modern Data Center*, accessed on Apr. 2017. [Online]. Available: http://images.nvidia.com/content/tesla/pdf/nvidia-teslap100-techoverview.pdf
[12] Y. Kim, J. Lee, and J. Kim, "GPUdmm: A high-performance and memory-oblivious GPU architecture using dynamic memory management," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, Feb. 2014, pp. 546–557.
[13] J. Lee, M. Samadi, and S. Mahlke, "VAST: The illusion of a large memory space for GPUs," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, 2014, pp. 443–454.
[14] S. Shahar, S. Bergman, and M. Silberstein, "ActivePointers: A case for software address translation on GPUs," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 596–608.
[15] F. Ji, H. Lin, and X. Ma, "RSVM: A region-based software virtual memory for GPU," in *Proc. Int. Conf. Parallel Archit. Compilation Techn. (PACT)*, 2013, pp. 269–278.
[16] PloyBench. *The Polyhedral Benchmark Suite*, accessed on Apr. 2017. [Online]. Available: http://web.cs.ucla.edu/ pouchet/software/polybench/
[17] JGF. *The Java Grande Forum Benchmark Suite*, accessed on Apr. 2017. [Online]. Available: https://www.epcc.ed.ac.uk/research/computing/performancecharacterisationandbenchmarking/javagrandebenchmarksuite
[18] *What is Aparapi?* accessed on Apr. 2017. [Online]. Available: http://aparapi.github.io/
[19] W. Zaremba, Y. Lin, and V. Grover, "JaBEE: Framework for object-oriented Java bytecode compilation and execution on graphics processor units," in *Proc. Workshop General Purpose Process. Graph. Process. Units (GPGPU)*, 2012, pp. 74–83.
[20] I. Gelado, J. E. Stone, J. Cabezas, S. Patel, N. Navarro, and W.-M. W. Hwu, "An asymmetric distributed shared memory model for heterogeneous parallel systems," in *Proc. Archit. Support Program. Lang. Oper. Syst. (ASPLOS)*, 2010, pp. 347–358.
[21] E. Holk, R. Newton, J. Siek, and A. Lumsdaine, "Region-based memory management for GPU programming languages: Enabling rich data structures on a spartan host," in *Proc. Int. Conf. Object Oriented Program. Syst. Lang. Appl. (OOPSLA)*, 2014, pp. 141–155.

BONGSUK KO received the B.S. degree in electrical engineering from the Gwangju Institute of Science and Technology in 2015, where he is currently a Graduate Student with the School of Electrical Engineering and Computer Science.



SEUNGHUN HAN received the B.E. degree in computer science and engineering from Chonbuk National University, in 2015. He is currently a Graduate Student with the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology.

YONGJUN PARK received the Ph.D. degree in electrical engineering from the University of Michigan, Ann Arbor, MI, USA, in 2013. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Hanyang University, Seoul, South Korea. His research interests include compilers and computer architectures for various computer systems.

MOONGU JEON received the B.S. degree in architectural engineering from Korea University in 1998, the M.S. degrees in computer science and civil engineering from the University of Minnesota at Mineapolis, and the Ph.D. degree in scientific computation, in 1994, 1999, and 2001, respectively. He is currently a Professor with the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology. His research interests include optimization and pattern recognition.

BYEONGCHEOL LEE (M'17) received the B.E. degrees in electronic and electrical engineering and computer science engineering from POSTECH in 2004, and the M.A. and Ph.D. degrees in computer science from the University of Texas at Austin in 2006 and 2011, respectively. He is currently a Full-Time Instructor with the Gwangju Institute of Science and Technology. His research interests include dynamic analysis and optimization.