2014 IEEE 28th International Parallel & Distributed Processing Symposium

BigKernel — High Performance CPU-GPU Communication Pipelining for Big Data-style Applications

Reza Mokhtari and Michael Stumm
Department of Electrical and Computer Engineering
University of Toronto
Toronto, Canada
{mokhtari, stumm}@eecg.toronto.edu

***Abstract*—GPUs offer an order of magnitude higher compute power and memory bandwidth than CPUs. GPUs therefore might appear to be well suited to accelerate computations that operate on voluminous data sets in independent ways; e.g., for transformations, filtering, aggregation, partitioning or other "Big Data" style processing. Yet experience indicates that it is difficult, and often error-prone, to write GPGPU programs which efficiently process data that does not fit in GPU memory, partly because of the intricacies of GPU hardware architecture and programming models, and partly because of the limited bandwidth available between GPUs and CPUs.**

In this paper, we propose *BigKernel*, a scheme that provides pseudo-virtual memory to GPU applications and is implemented using a 4-stage pipeline with automated prefetching to (i) optimize CPU-GPU communication and (ii) optimize GPU memory accesses. BigKernel simplifies the programming model by allowing programmers to write kernels using arbitrarily large data structures that can be partitioned into segments where each segment is operated on independently; these kernels are transformed into BigKernel using straight-forward compiler transformations.

Our evaluation on six data-intensive benchmarks shows that BigKernel achieves an average speedup of 1.7 over state-of-the-art double-buffering techniques and an average speedup of 3.0 over corresponding multi-threaded CPU implementations.

***Keywords*-GPU, CPU, communication, management, optimization, stream processing**

I. Introduction

An important class of computations operate on voluminous data sets in ways similar to what is sometimes referred to as "Big Data" computations and other times referred to as *streaming computations*. These computations perform simple, straightforward, and independent operations on a large number of input data records, one chunk at a time, and hence are trivially parallelizable. This is a large class and includes computations that filter, transform, aggregate or partition large data sets. We are interested in using GPUs for these types of computations.

On the surface it appears that GPUs would be ideal for this. The many GPU cores allow for highly parallelized processing and offer large aggregate compute power compared to CPUs; e.g., 4.5 TFLOPS vs. 460 GFLOPS (Nvidia GTX Titan vs. Intel Ivy Bridge). And GPU memory has significantly higher theoretical bandwidth than CPU memory, since they were designed for graphics processing; e.g., 288 GB/s vs. 52 GB/s.

However, a number of issues complicate the efficient use of GPUs for these types of computations. Firstly, CPUs and GPUs have separate memories, requiring explicit data transfers between CPU and GPU memory, and GPU memory is limited in size (currently up to at most 6GB). Because of the limited size, the large quantity of data to be processed needs to be explicitly partitioned into chucks and iteratively copied into GPU memory for processing there. Efficient partitioning is not always straight-forward, especially when dealing with (non-indexed) variable-length records. To enable concurrency between GPU processing and data transfers, double-buffering is typically used where the GPU processes the data in one of the buffers while data is being transferred into the other buffer. But coding double buffering has been shown to be tedious and error-prone [12].

Secondly, the PCIe link that connects the two memories has limited bandwidth and, for the computations we are considering, can often be a bottleneck, starving GPU cores from their data. For example, PCIe Gen 3 has a theoretical maximum throughput of 15.75 GB/s, far lower than memory bandwidth GPU-side, and difficult to exploit in practice. Indeed, while impressive speedups have been reported for many GPU applications, the speedups were often calculated without taking into account the overhead of transferring data between CPU and GPU memory [6].

Thirdly, the high bandwidth of GPU memory can only be exploited when GPU threads executing at the same time access memory in a *coalesced* fashion, where the threads simultaneously access adjacent memory locations; otherwise memory accesses may become serialized, resulting in significantly lower memory throughput. Low memory throughput for data-intensive applications, like the ones we are targeting, results in highly underutilized GPU cores. Because such applications are not a priori structured to operate on data in a coalesced fashion, the data has to be reorganized to support coalesced accesses on the GPU, which is non-trivial.

In this paper, we describe *BigKernel*, a mixed compile-time, run-time scheme that addresses the issues listed above by using data prefetching within a 4-stage pipeline. The key idea behind BigKernel is to have GPU threads identify online, but ahead of time, which data they will access in their computations, transfer this information to the CPU, and then have the CPU assemble the data and transfer it to GPU memory prior to when the GPU threads access the data.

This scheme has a number of potential advantages. First, the amount of data transferred over the PCIe link from CPU

1530-2075/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPS.2014.89

memory to GPU memory is often reduced, because only the data being accessed GPU-side is transferred (as opposed to all data). Secondly, it allows the CPU to assemble the data in a way that increases coalesced data accesses on the GPU for more efficient GPU memory accesses. Finally, it significantly simplifies the programming model, since the programmer need not deal with and manage (i) buffers, (ii) the transfers of data between CPU and GPU, and (iii) the reorganization of data so as to enable coalesced accesses.

BigKernel allows the programmer to write a GPU program with arbitrarily large data structures as if (pseudo-) virtual memory were available GPU side, if they are accessed in a streaming fashion. The program is compiler-transformed to one that automates the management of buffers, the transfers, and the layout of GPU-side data in a way that is transparent to the programmer. Moreover, the fact that the transformed program only invokes a single kernel once for the entire computation means that the kernel context (i.e., registers and GPU shared memory) is available throughout the entire computation without having to manage it separately as would be the case when kernels are invoked iteratively, further simplifying her programming efforts.

However, one should note that BigKernel is not a general framework. It targets only a subset of computations, namely streaming computations, which are computations that operate on input records in independent ways. However, we argue that this is a large and important subset. Moreover, BigKernel involves a number of tradeoffs. For example, BigKernel uses twice as many GPU threads, potentially limiting the degree of parallelism GPU-side, although we have found GPU core utilization to be low for the class of computations being considered. As another example, BigKernel uses more CPU-side resources compared to traditional schemes, i.e., more CPU threads, more memory accesses, and more buffers that are pinned so that they cannot be paged out, which may impact other concurrently running processes on the CPU.

Our initial performance evaluation, presented in Section VI, is encouraging. With six benchmarks, we show that BigKernel outperforms corresponding single and double buffering implementations across all benchmarks by up to 4.6X and 3.1X and on average by 2.6X and 1.7X, respectively. Compared to corresponding multi-threaded CPU implementations, BigKernel executes up to 7.2 times faster and on average 3.0 times faster.

As far as we know, BigKernel[1] is:

1) the first scheme to improve on the performance of state-of-the-art double-buffering schemes for GPUs;
2) the first scheme to automate CPU-GPU data transfers for large data sets without requiring the programmer to split the data or annotate the code;
3) the first scheme to provide the continuous execution of a single kernel on arbitrarily large input/output data sets;

Section II gives a brief background on GPUs for those less familiar with GPUs. Sections III and IV describe BigKernel in more detail. Sections V and VI present our performance evaluation. We close with related work and concluding remarks.

[1]The name *BigKernel* was chosen because (i) its target applications are those with Big Data-style processing of large datasets, (ii) the programmer can write a single "big" kernel that can operate on all data, even if the data does not fit in memory, and (iii) the kernel that is generated is big compared to a traditionally implemented kernel; e.g., a kernel that is implemented in 70 LOC is transformed into one that has over 500 LOC.

II. GPU Background

GPU hardware consists of several *streaming multiprocessors* (SM), each of which contains multiple computing cores, registers, and a small (e.g. 64KB) but fast on-chip memory called *shared memory*, accessible to the cores of the SM. The cores of an SM execute in lock-step with each core executing the same instruction at the same time; i.e., in SIMD fashion.

The GPU is connected to an off-chip DRAM memory called *global memory*, with up to a few gigabytes of space to all cores of all SMs. Accesses to global memory is an order of magnitude slower than accesses to registers and shared memory. We refer to this global memory as *GPU memory* in this paper to differentiate it from CPU main memory.

The GPU is connected to the CPU through a bidirectional link, such as PCIe. The GPU will have one or more DMA engines capable of transferring data over this link between CPU and GPU memory. However, the DMA engine can only access CPU memory that has been pinned so that it will not be paged out by the OS. The DMA engines also provide GPU cores direct (but slow) access to CPU memory.

The term *kernel* is used to denote the function that executes on the GPU by a collection of threads in parallel. The programmer configures the kernel to be executed by a given number of GPU threads. These threads are grouped into *thread blocks* as configured by the programmer. Each thread block is assigned to a SM by a hardware scheduler. Threads of a thread block are further divided into groups of 32, called *warps*. The threads in a warp execute in lock-step because the cores share the same instruction scheduler. *Thread divergence* will occur if, on a conditional branch, threads of the same warp take different paths, which can lead to serious performance degradations. Inter-warp divergence does not negatively impact performance.

III. BigKernel Overview

BigKernel organizes a computation into four pipeline stages:

1) **Prefetch address generation** (GPU-side): GPU threads calculate the addresses of the data needed by the computation threads later in Stage 4. The code to generate the memory addresses is obtained from the original GPU kernel source code by removing all statements except (i) those that contribute to control flow, (ii) those that contribute to memory access address calculations, and (iii) the memory accesses themselves. The memory access instructions are transformed so that instead of making memory accesses, the address is recorded in an *address buffer* CPU-side reserved for that purpose (See Fig. 1).

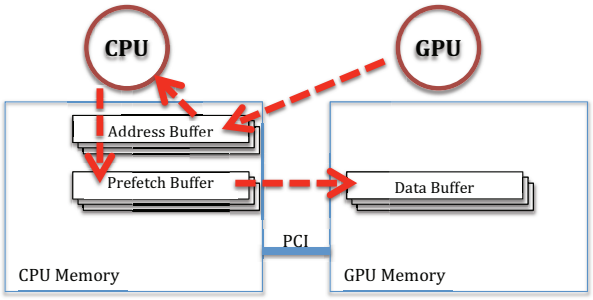


Fig. 1. BigKernel buffers

2) **Data assembly** (CPU-side): The prefetch data is assembled into a *prefetch buffer* in CPU memory based on the addresses generated in Stage 1.
3) **Data transfer**: the GPU DMA engine transfers the contents of the prefetch buffer to a *data buffer* in GPU memory.
4) **Kernel computation** (GPU-side): GPU threads execute the actual computation using the prefetched data. The code for the computation is obtained by transforming the original GPU kernel to use the prefetched data in the data buffer instead of the original memory accesses.

A. A simple example

To provide more detail using an example, consider part of a K-means computation:

GPU-side code
```
clusterKernel(particles, numP, clusters) {
   for( i = 0; i < numP; i++)
      particles[i].cid = findClosestCluster(
               particles[i].x, particles[i].y,
               particles[i].z, clusters);
}
```

where `particles` represents a `numP` element particle array that does not fit in GPU memory but is accessed in a streaming fashion, `clusters` represents an array of clusters that fits entirely in GPU memory, and `findClosestCluster()` returns the id of the cluster closest to the target particle.

Because the particle array does not fit in GPU memory, the array would traditionally be processed in chunks with the following GPU code invoked iteratively (disregarding `findClosestCluster()`'s return value for now to simplify the example):

GPU-side code
```
clusterKernel(particles, numP, clusters) {
   start = myParticleStartIndex( threadId, numP );
   end = myParticleEndIndex( threadId, numP );
   for(i = start; i < end; i++)
      findClosestCluster(particles[i].x,
      particles[i].y, particles[i].z, clusters);
}
```

The corresponding CPU code (i) allocates space for the cluster array GPU-side and copies the cluster array to GPU memory, and then (ii) iteratively copies the next chunk of the particles array to GPU memory before invoking the GPU `clusterKernel()` on the chunk:

CPU-side code
```
cudaMalloc(d_clusters, clSize);
cudaMemcpy(d_clusters, clusters, clSize);
cudaMalloc(d_pBuf, GPUBufSize);
for(offset=0;offset<pSize;offset+=GPUBufSize) {
  cudaMemcpy(d_pBuf, particles + offset, GPUBufSize);
  clusterKernel<<<>>>(d_pBuf, pElements, d_clusters);
}
```

(`pElements` is equal to the number of particles that fit in the GPU buffer `pBuf`.)

Using BigKernel, the programmer no longer needs to partition the large particle array into chunks and manage the transfers. Instead, she would provide the following CPU-side code that assumes the existence of an arbitrarily large `d_particles` array in GPU memory that is (virtually) allocated with `streamingMalloc()` and mapped to the CPU-side particles array `particles` with `streamingMap()`:

CPU-side code
```
cudaMalloc(d_clusters, clArraySize);
cudaMemcpy(d_clusters, clusters, clArraySize);
streamingMalloc(d_particles, pArraySize);
streamingMap(d_particles, particles, pArraySize);
clusterKernel<<<>>>(d_particles, numP, clusters);
```

The corresponding GPU-side kernel code written by the programmer remains unchanged but gets invoked only once.

Note that the K-means kernel accesses two types of data structures: the cluster array which is explicitly copied to GPU memory and does not involve BigKernel, and the particle array that does not fit in GPU memory and is mapped. BigKernel manages the accesses to the latter.

We now describe each of the four stages of the BigKernel pipeline in more detail, using the same running example.

Prefetch address generation: The prefetch address generation code is obtained from the original GPU kernel by transforming the read accesses to the `d_particles` array in the for-loop to instead store the addresses in an `addrBuf` array CPU-side:

GPU-side code
```
counter = 0;
for( i = start; i < end; i++ ) {
   addrBuf[counter++][threadId] = &particles[i].x;
   addrBuf[counter++][threadId] = &particles[i].y;
   addrBuf[counter++][threadId] = &particles[i].z;
}
```

Currently, our transformations are relatively simplistic in that they cannot deal with indirections or flow control based on application data that may be modified, in which case, then the transformation simply defaults to fetching all data, making the resulting code similar to the double-buffering scheme.

This technique has three potential downsides. First, if the same data item is accessed multiple times in the code then it will be be transferred multiple times, leading to extra overhead. However, we have not found this to be the case with the applications we examined, and believe that it is

rare in Big Data-style, streaming applications. Second, when characters (which are typically 1-byte) are accessed then the communication overhead of transferring the addresses (which are typically 4 or 8-bytes) is far greater than the overhead of transferring the characters. We address this issue in the next section. Finally, the CPU-side address buffer must be pinned (i.e., non-pageable) so that it can be accessed by the GPU DMA engine. While this consumes physical CPU memory that cannot be made available to other processes, this should rarely become an issue given today's CPU memory sizes.

Data assembly: A dedicated CPU thread is responsible for fetching the corresponding data element from the particles array for each address in the address buffer and placing it in the prefetch buffer, which also must be a pinned buffer.

Note that the layout in the prefetch buffer enables coalesced accesses after it has been transferred to GPU memory. This is because the addresses were stored into the address buffer in the order they were accessed by the GPU threads, so data accessed at the same time will be adjacent to each other.

This data assembly process has one potential disadvantage. Because data assembly occurs CPU-side, it involves twice as many memory accesses CPU-side compared to traditional GPGPU applications. In traditional GPGPU applications, data for the GPU is first copied to a pinned buffer, resulting in a CPU-side read and a write for each data element. However with BigKernel, the address is first DMAed to memory by the GPU, the CPU then reads the address before copying the target data to the pinned prefetch buffer, resulting in two reads and two writes for each prefetched data element.

Data transfer: The data transfer stage is executed by the GPU DMA engine, allowing CPU and GPU cores to concurrently do other work.

Kernel computation: The computation code is (compiler-) generated from the GPU kernel by transforming the accesses to the `particles` array in the kernel `for`-loop to instead access the data previously transferred from the CPU in `dataBuf`:

GPU-side code

```
counter = 0;
for( i=start; i < end; i ++ )
   findClosestCluster (
            dataBuf[counter++][threadId],
            dataBuf[counter++][threadId],
            dataBuf[counter++][threadId],
            clusters
            );
```

Four-stage pipeline: Fig. 3 shows the implementation of the 4-stage pipeline depicted in Fig. 2, assuming a single thread block. Unlike what is shown in Fig. 2, the time it takes for each stage to complete in practice will vary, depending on the application. However, prefetch address generation takes the least amount of time across all applications we experimented with.

The CPU launches twice as many GPU threads as specified in the original program. Half are responsible for generating

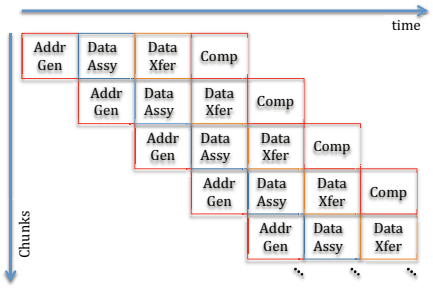


Fig. 2. Four-stage pipeline

the prefetch addresses and the other half are responsible for the computation. The GPU threads must be launched in a way such that each warp contains only address generation threads or only computation threads, but not both — otherwise the kernel will suffer from thread divergence.

The outermost `for`-loop of the kernel (lines 10-38) processes one chunk of data at a time. The address generation stage (lines 13-20) ends when `addrBuf` is full, at which time all the address generating threads first barrier (line 22) and one of the threads in thread block signals the CPU that the addresses are ready. The CPU waits until the addresses are ready (line 55) and then assembles the data (lines 56-57), copies the data to the GPU (line 59) and then signals the GPU computation threads that the data is ready (line 60), at which time the computation stage (lines 26-37) can commence.

Some details were omitted from the code in Fig. 3 for simplicity. For example, multiple instances of each buffer are required to allow for concurrency (although the code only shows one). At minimum, two of each are required so that one can be produced while the other is still being consumed.

Writes to mapped data: Writes to steaming data are handled similar to the way reads are handled: each write results in the writing of the target address to an address buffer CPU side and the data value is written to a write buffer GPU side, which is then transferred to CPU memory by the DMA engine. This requires two extra set of buffers: one GPU side to collect the writes, and one CPU side to which the data is transferred. This also adds two stages to the pipeline: one for the data transfer back to CPU memory and one for a CPU thread to process the transferred data and update the target data structure.

Multiple GPU thread blocks: The examples and code above assumed all threads were running within one GPU thread block. Supporting multiple thread blocks adds a few complications which, however, are handled through compiler transformations in a straightforward way. A separate set of buffers is needed for each thread block both CPU- and GPU-side. A separate CPU thread for each GPU thread block is responsible for data assembly CPU-side. Threads within a thread block need to be organized so that half of them are responsible for prefetch address generation and the other half are responsible for computation so that each computation

GPU Side:

```
clusterKernel(particle, numP, clusters,
                    addrBuf, addrBufSize
                    dataBuf, dataBufSize)
{
  tid = getVirtualThreadId(threadId);
      // 1 addrGen and 1 comp thread assigned same tid
  start = myParticleStartIndex(tid, numP);
  end = myParticleEndIndex(tid, numP);

  i = start;
  for(; i < end;) //each iteration: processing one chunk
  {
    if(isAddrGenThread(threadId))
    {
      counter = 0;
      for( ; (i < end) && (counter < addrBufSize); i ++)
      {
        addrBuf[counter ++][tid] = &particle[i].x;
        addrBuf[counter ++][tid] = &particle[i].y;
        addrBuf[counter ++][tid] = &particle[i].z;
      }

      barrier_addrGenThreads() ;
      signal_addrReady() ;
    }
    else // computation thread
    {
      counter = 0 ;
      wait_dataReady() ;

      for( ; (i < end) && (counter < dataBufSize); i ++)
      {
        findClosestCluster(dataBuf[counter ++][tid],
                           dataBuf[counter ++][tid],
                           dataBuf[counter ++][tid]
                          );
      }
    }
  }
}
```

CPU Side:

```
cudaMalloc(d_clusters, clArraySize);
cudaMemcpy(d_clusters, clusters, clArraySize);

cudaMalloc(d_dataBuf, dataBufSize);
pinnedMalloc(h_addrBuf, addrBufSize);
pinnedMalloc(h_pBuf, pBufSize);

clusterKernel<<<>>>(d_particle, numP,
                        d_clusters, numCl,
                        h_addrBuf, addrBufSize,
                        d_dataBuf, dataBufSize
                       );

while(GPUKernelisRunning())
{
  wait_addrReady();
  for(i = 0; i < numAddr; i++)
        h_pBuf[i] = *(particles + addrBuf[i]);

  cudaMemcpyAsync(d_dataBuf, h_pBuf);
  signal_dataReady() ;
}
```

Fig. 3. Implementation of the 4-stage pipeline.

thread can run in the execution context of the corresponding address generation thread.

IV. Optimizations

A. Pattern recognition

The address generation stage makes use of a pattern recognition component that attempts to extract patterns from the memory addresses it generates. The goal of this component is to reduce the amount of address information that needs to be sent to the CPU by replacing the generated addresses with a pattern that can be used to reproduce the addresses. Such pattern recognition, if successful, is particularly impactful performance-wise when dealing with text-based input data, since an address (4 or 8-bytes) would otherwise be required for each accessed character (1-byte).

Each address generation thread starts by generating a few addresses, storing them in a private temporary address buffer.[2] The number of addresses generated is dictated by the size of the buffer, which is typically a few tens of bytes. It then invokes a pattern recognition function to identify a potential pattern from the stored addresses. A pattern, if found, consists of a base address and a number of strides between subsequent addresses. For instance, if stored addresses are 0x00100, 0x00105, 0x00110, 0x00115, then the pattern would be *[base_address: 0x00100, stride(s): 5]*. If no pattern is found, then the addresses collected in the temporary buffer are copied to the CPU-side address buffer, and address generation continues as described earlier in Section III.

If a pattern is identified, then the address generation thread continues generating data access addresses, but now verifies that each subsequently generated address follows the identified pattern. If it does not, then address generation is started again, this time without attempting to identify a pattern and writing the addresses to the CPU-side address buffer.

If all subsequently generated addresses adhere to the pattern, then the pattern (instead of the addresses) is written to CPU memory, and a signal is sent to the CPU indicating that a pattern was found.

The pattern recognition scheme described above is rather simplistic, but we have found it to be effective with our benchmarks — see Section VI. One can easily conceive of ways to extend it and make it more versatile (e.g., allow patterns to change midstream).

B. Data locality in assembling data

Compared to traditional double-buffering implementations, BigKernel incurs extra memory accesses during the data assembly stage CPU-side. If access patterns are provided by the prefetch address generation stage then the overhead of the data assembly can be reduced by improving memory access locality and the attendant cache hit rate.

We focus on improving memory access locality when reading data from the source of the mapped data, as opposed to when writing data to the prefetch data buffer, because we found that the cost of these reads is far higher than the cost of the writes.

To read the prefetch data specified by the pattern, instead of reading the data items in the order they are needed by the GPU computation threads, we read all of the prefetch data for one GPU thread at a time. This results in increased data locality in CPU reads, because each GPU thread tends to access consecutive data. The fetched data is, however, still stored in the prefetch data buffer in the order they will be

[2]Preferably these temporary buffers are allocated in GPU shared memory, but if there is not enough space there because it is needed by the computation, the buffer is allocated in GPU memory.

accessed GPU-side. If multiple data structures are mapped and accessed by the GPU, then we additionally read the data from each structure separately.

C. Synchronization

Synchronization in GPGPU applications is complicated by the intricacies of the GPU hardware. In particular, there is no signaling mechanism between CPU and GPU beyond using flags located in memory and busy waiting for a specific flag value. For this reason, it is important to implement synchronization so as to minimize the number of memory accesses required, especially on the GPU because of the large number of threads that execute there.

The first three stages of the BigKernel pipeline are producers for their following stages: the address generation stage produces addresses for the data assembly stage, which produces data for data transfer stage, which produces data for computation stage. For each buffer used in the pipeline, proper synchronization is required to ensure that consumption of the buffer data does not commence before the data has been produced, and that data for a buffer is not produced until the buffer has previously been consumed.

The GPU signals the CPU at the end of the address generation stage by setting a flag in CPU memory. The CPU busy waits on that flag before it starts the data assembly stage. The GPU cannot signal the CPU until all of the address generating threads have completed their stage. Hence, the address generation threads first barrier at the end of their stage before one of the threads signals the CPU. We use the `bar.red` GPU instruction for barriering, because it is efficient and can barrier a given number of threads.

No synchronization is needed between the data assembly stage and the data transfer stage, because the latter is initiated by the CPU thread after it completes the data assembly.

The DMA engine knows when the data transfer stage has completed, but there is no mechanism for the DMA engine to signal the kernel computation threads that this occurred. We rely on the fact that the DMA engine performs data transfers in order. After the CPU instructs the GPU DMA engine to transfer the data buffer (using *cudaMemcpyAsync*), it instructs the DMA engine to copy a flag to a specific location in GPU memory that indicates the data transfer has completed. The flag will not be transferred until the data buffer has been transferred.

Instead of having each GPU computation thread busy wait on that flag, only one computation thread is assigned that task. All the other computation threads barrier (again using `bar.red`) to ensure they do not start the computation phase until the flag has been set.

To prevent subsequent address generation stages from overwriting an address buffer that has not yet been consumed, we barrier all threads in a thread block once for each chunk iteration.[3] Each address generation thread in iteration n synchronizes with the computation threads in iteration $n-3$. This relies on the fact that when a computation stage starts, all three stages prior to it have completed and the buffers of the previous stages can safely be overwritten.

Synchronization between threads across different thread-blocks is not needed because both computation threads and their corresponding address generation threads are packed into the same thread-block and they interact with a separate CPU thread responsible for their data prefetching.

D. Buffer allocation: active vs. inactive thread-blocks

To ensure efficient use of memory resources both CPU- and GPU-side, BigKernel allocates data and address buffers only for active thread-blocks, reusing them when inactive thread-blocks become active[4] (which only occurs when a resident active thread-block retires). The benefit of allocating buffers only for active thread-blocks is that buffers can be made larger, potentially improving performance by reducing the number of synchronization points.

We use a hybrid compile-time, runtime method to identify the number of active thread-blocks. First, the resource usage required by a thread block, R_{tb}, is determined at compile-time and provided as a constant value in the application's code. The resources provided by the GPU hardware, R_{GPU}, is then probed at runtime (using provided API functions). The number of active thread-blocks is then calculated as: $min(numSetBlocks, (R_{tb}/R_{GPU}))$ where $numSetBlocks$ is the number of thread-blocks set by the programmer as the argument of the kernel invocation.

V. Experimental Setup

Our baseline hardware infrastructure consists of a 3.8GHz Intel Xeon Quad Core E5 with 8 hardware threads and 10MB of combined L2/L3 cache, connected to 16GB of quad-channel memory clocked at 1800MHz. All GPU kernels were executed on an NVIDIA GeForce GTX 680 GPU with 1,536 computing cores each running at 1020MHz connected to 2GB of GPU memory. The GPU video card is connected to the system with a PCIe Gen3 x16 link interconnect.

All GPU-based applications were implemented in CUDA, using CUDA and GPU driver release 5.0.35 installed on a 64-bit Ubuntu 12.04 Linux with kernel 3.5.0-23. All applications are compiled with the corresponding version of the *nvcc* compiler using optimization level three.

For our experiments, we ran six applications with a range of different data access patterns:[5]

K-means: partitions n particles into k clusters so that particles are assigned to the cluster with the nearest mean. Each *particle* consists of the particle's coordinates, their clusterIds, and a few other data values. The kernel reads particle coordinates and sets its clusterId; the clusterIds therefore have to be transferred back to CPU memory.

[3] An alternative is to use full/empty flags for each buffer, but this increases the number of data transfers and the amount of busy waiting.

[4] The number of thread-blocks that become active depends on the resources (i.e. registers and shared memory) that each thread-block requires and the total resources provided by the GPU.

[5] The source code for these applications as well as their input data is available at *http://www.eecg.toronto.edu/~mokhtari/bigkernel*.

Application	Data Size	Record Type	Mapped Data Access Proportion	
			Read	Modified
K-means	6.0GB	Fixed-length	50%	12%
Word Count	4.5GB	Variable-length	100%	0%
Netflix	6.6GB	Fixed-length	30%	0%
Opinion Finder	6.2GB	Fixed-length	73%	0%
DNA Assembly	4.5GB	Fixed-length	36%	0%
MasterCard Affinity	6.4GB	Variable-length	100%	0%
MasterCard Affinity (indexed)	6.4GB	Variable-length (indexed)	25%	0%

TABLE I
APPLICATION MAPPED DATA DATA. (AN APPLICATION MAY ALSO ALLOCATE AND ACCESS OTHER NON-MAPPED DATA STRUCTURES.)

Word Count: counts the number of occurrences of each word in a large, mapped document.

Netflix: predicts user preferences of movies [3]. An array of records consisting of movie user ratings and a few other data values is mapped. The user ratings of each pair of users to a movie is read from a record and the correlation between the two ratings is stored in a GPU-side pre-allocated table.

Opinion Finder: analyzes the sentiments of tweets associated with a given subject (i.e. a set of given keywords) [17]. The mapped data consists of a set of records that each include a tweet, a time-stamp, and a few other data values. Words from each tweet that mention the given subject are looked up in three dictionaries of positive, negative, and adverb words. Based on the identified words and their precedence, an overall sentiment score is calculated. The dictionaries are not mapped. The output is an aggregated value that represents the sentiment score of the tweets.

DNA Assembly: merges fragments of a DNA sequence to reconstruct a larger sequence [2]. An array of records, each consisting of a fixed-length DNA fragment, a string value, and a few other data values, is mapped. For each fragment, the application hashes a portion of the fragment and stores it in a hash table to count the number of identical fragments and to remove the noisy ones. The hash table is later used to incrementally extend each fragment by finding partial overlaps between different fragments.

MasterCard Affinity: finds all merchants that are frequently visited by customers of a target merchant X. A collection of purchase transactions is mapped, where each transaction includes credit card number, the payment terminal ID, and several other values. The application first extracts a list of customers that visited merchant X and then, with another pass over the purchase transactions, identifies the merchants visited by the customers from the list. The output is a table of merchants visited by all customers of merchant X, along with frequency information.

MasterCard Affinity (indexed): as above, except that an extra index file is provided that contains offsets to the data-fields within the input.

Table I provides more details on the application data sets and how they are accessed. Applications that do not modify mapped data, write results to GPU memory that is transferred to CPU memory after all computations have completed.

VI. EXPERIMENTAL RESULTS

To evaluate BigKernel, we implemented five different variations of our applications: (i) a CPU-based serial implementation, (ii) a CPU-based multi-threaded implementation, (iii) a GPU-based implementation that uses a single buffer for data transfers, thus serializing computation and data communication, (iv) a GPU-based implementation that uses double-buffering for data transfers in order to overlap computation with data communication, and (v) BigKernel.

All GPU-based implementations use the same kernel. Each implementation is configured to run with the number of GPU computation threads that results in the best execution time, as determined through experimentation. Moreover, each implementation uses buffer sizes that result in the best execution time, given memory constraints.

The performance results presented here represent the average over ten consecutive runs.

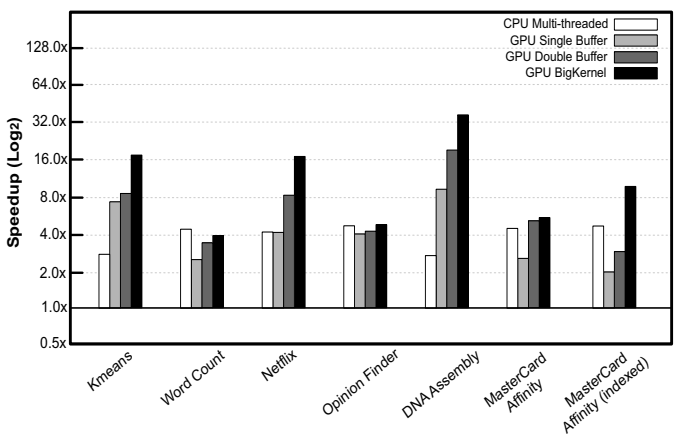
A. Overall results

Fig. 4(a) depicts the speedup of all implementations relative to the CPU-based serial implementation. To help interpret the speedups for the GPU-based implementations, Fig. 4(b) shows the computation / communication ratio of the single-buffer implementation.

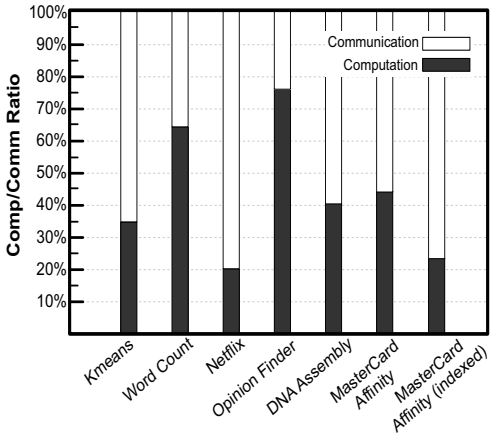
BigKernel outperforms both the single and double-buffering implementations across all applications. The performance gains can primarily be attributed to (i) overlapped computation and data communications, (ii) reducing the volume of CPU-GPU data communications and, (iii) enabling coalesced accesses to GPU memory by placing the input data of consecutive threads in interleaved data segments.

Word Count and Opinion Finder have a relatively low speedup and do not appear to benefit from optimized CPU-GPU data communications, primarily because they have a dominant computation stage, as we show further below, which prevents improvements from overlapping computation with communication or from data transfer reductions. Word Count uses a centralized hash table to store word counts, requiring synchronization with attendant overheads. Opinion Finder's computation is dominant because of the fairly heavy lexical analysis it conducts on input tweets.

The speedup of MasterCard Affinity is also limited due to the fact that entire input dataset has to be transferred to GPU memory, because the variable-length records force the computation to go over all of the data to identify the individual records (which are delimiter-character separated). The small performance advantage of BigKernel over the double-buffering version is due to the effect of memory coalescing. The indexed version of MasterCard Affinity, however, achieves significant speedup, because it reduces the amount of data transferred, and because the benefits of coalesced memory accesses become more exposed with the more efficient data transfer stage.



(a) Application speedup over serial CPU implementation.



(b) Comp/comm ratio in single-buffer implementation.

Fig. 4. Overall performance results.

B. Performance breakdown

To gain more insight into which features of BigKernel lead to performance improvements, we ran BigKernel with certain features disabled and measured the speedup obtained over the single-buffered implementation:

1) **BigKernel overlap only**: this variant transfers all data in its original layout; i.e., no optimizations to reduce the data transferred and no optimization for increased coalesced accesses. Hence, this variant thus only provides pipelined execution, where communication and computation is overlapped.
2) **BigKernel transfer volume reduction**: this variant transfers only the data required by the computation but leaves the transferred data in its original layout (with the optimizations for coalesced accesses disabled).
3) **BigKernel**: the complete BigKernel implementation.

Fig. 5 depicts the incremental speedup obtained from running one variant over the other. The figure thus gives an indication of the contribution of reduced data transfers and data layout optimized for coalesced accesses.[6]

As expected, the data transferred for MasterCard Affinity and Word Count cannot be reduced and therefore they only benefit from communication/computation overlap and memory coalescing. Opinion Finder also does not exhibit performance improvements from reducing the CPU-GPU data transfers due to its dominant computation stage.

The effect of the memory coalescing optimization varies from application to application based on a number of factors: 1) the ratio of accesses to mapped data over all data accesses in the kernel – the higher the ratio, the greater the performance benefit; 2) whether the data transfer stage dominates or not – if it does, there is no benefit from optimizing memory accesses; and 3) whether or not the original layout already leads to highly coalesced accesses – if so, there is not much room for improvement.

[6]It should be noted that the graph would look substantially different if the disabling of features had been done in a different order, because the contributions of each feature overlap in the pipeline.

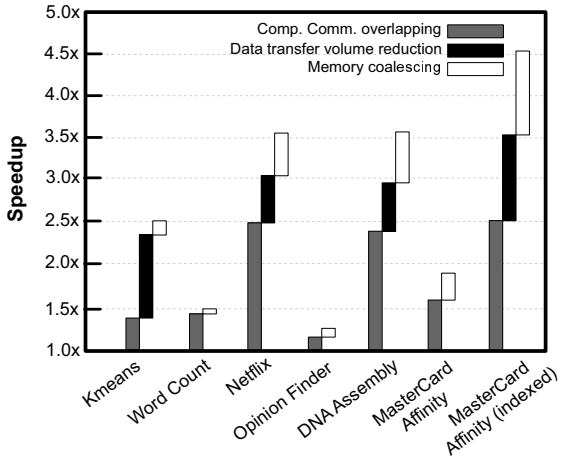


Fig. 5. The incremental benefit of (i) overlapping computation and communication, (ii) reducing the volume of data transferred due to prefetching, and (iii) laying out the data to increased coalesced accesses.

Note that many of our target applications are inherently incapable of exhibiting coalesced memory accesses in their original form. The records being processed are often large and therefore, only a few of them can be accessed in each memory transaction, causing the memory accesses of consecutive threads to be non-coalesced. This is the case, for instance, in our DNA assembly application where each DNA fragment record is typically so large that data from different records cannot be accessed within a coalesced memory transaction. Moreover, in applications with variable-length records, consecutive threads cannot be easily assigned to process consecutive records in an interleaved fashion because it is difficult for consecutive threads to identify the starting memory location of consecutive records without accessing the previous records.

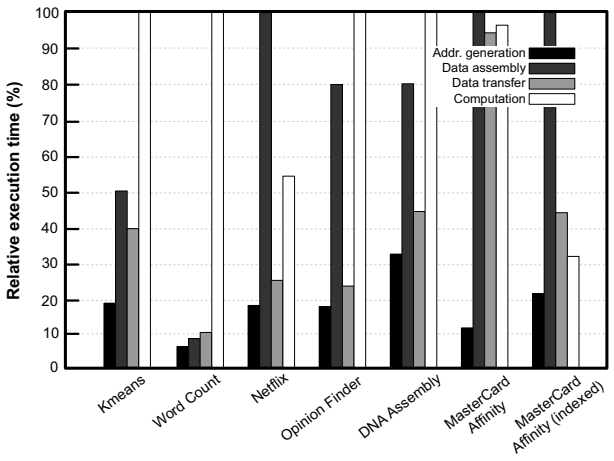


Fig. 6. Relative completion time of each BigKernel stage.

C. Stage completion time breakdown

For optimal execution, each stage in the BigKernel pipeline would ideally take the same amount of time to complete. This is obviously not the case, and the amount of time each stage requires to complete varies from application to application. For each application, we experimentally measured the time each stage required on average to complete.[7]

Fig. 6 shows, for each application, the time each stage took to complete on average relative to the stage that took the longest. The address generation stage requires only a small fraction of the total execution time (usually less than 20%) as it only executes those instructions that contribute to memory address calculations.

The time taken by the data assembly stage varies for the different applications based on (i) the amount of data that has to be assembled and (ii) the data locality of the data items being accessed by CPU memory and hence the cache hit rate.

The time taken for the data transfer solely depends on the size of data to be transferred because the data to be transferred is put in a contiguous pinned-buffer and therefore is efficiently transferred to GPU memory by the GPU DMA engine.

Finally, the computation itself is responsible for a considerable portion of the execution time. Clearly, this is expected for those applications that originally had a dominant computation stage. However, it is interesting to note that BigKernel significantly increased the computation / communication ratios relative to the original single-kernel implementations (Fig. 4(b)). The fact that the computation stage is the slowest stage for many of these applications indicates that GPU memory may be the bottleneck.

D. Pattern recognition

Recognition of access patterns during the prefetch address generation stage is a key optimization in BigKernel. This is shown in Table II that lists the performance improvements when only having to transfer patterns to CPU memory over having to send the actual addresses.

[7]To measure execution breakdown, we inserted time measurements at the beginning and end of each stage. The data transfer stage, in particular, is measured by having the CPU continuously ping the status of data transfers to stop the timer when the transfer has completed.

Application	Performance difference
K-means	31%
Word Count	66%
Netflix	3%
Opinion Finder	6%
DNA Assembly	7%
MasterCard Affinity	57%
MasterCard Affinity Indexed	NA

TABLE II
PERFORMANCE IMPROVEMENT DUE TO THE USE OF ACCESS PATTERNS.

The extent to which performance is improved for each application depends on the number of addresses sent during the address generation stage which in turn depends on the granularity of the data being accessed. For instance, in K-means, one address is sent for each *double* variable (i.e. 8-byte) while in Word Count, one address has to be sent for each required character (i.e. 1-byte). Having to send a large number of addresses relative to the amount of data to be transferred adds significant overhead to PCIe transfers (for addresses) and on CPU memory (to read addresses during the data assembly stage). Replacing the addresses with a pattern can thus have a significant performance impact.

VII. Related Work

Managing CPU-GPU data communications is a well-known challenge both from a programmability and a performance point of view. The majority of existing work addresses this challenge only from a programmability point of view [5], [8], [13], [18]. Some prior work also considers performance, which is more closely related to our work.

Komoda et al. propose a library for OpenCL that automatically overlaps computation with data communication given the memory usage pattern of a kernel [12]. The performance of the resulting applications is close to that of the double-buffering scheme. However, the programmer is still required to provide various details on the data usage pattern of the kernel.

CGCM and DyManD are two systems that automate CPU-GPU data communications through a hybrid compile-time and run-time scheme [10], [11]. However, the data transfers are not overlapped with computation. And the programmer is still responsible for splitting the data into smaller chunks and invoking the kernel multiple times.

Pai et al. propose a system that automates CPU-GPU memory management based on a coherence scheme in order to reduce superfluous communication [14]. To do this, when a data item is accessed on one side (CPU or GPU side), it is transferred (from the other side) if it is not locally available or if its local version is stale. This system does not overlap computation and communication.

There is interesting prior work to automatically manage and optimize data transfers, but they target transfers between GPU global memory and GPU shared memory. [1], [7], [9]. Interestingly, they also target streaming computations, but

they assume the data is already in GPU global memory. CUDA-lite translates an annotated kernel so that it prefetches the data from GPU memory in a coalesced fashion and stores it in GPU shared memory, to where future data accesses are redirected [16]. In a similar work, Yang et al. proposes various compiler optimizations, including one that converts non-coalesced accesses into coalesced ones through the use of shared memory [19]. None of this prior work considers CPU-GPU data communications.

Our prefetching scheme is related to a method of dealing with irregular problems in distributed systems known as *inspector-executor* [4], [15].

Finally, Zhang et al. optimize the layout of irregularly accessed data to achieve more efficient GPU memory accesses by having the CPU place elements of an irregularly accessed array that will be accessed by GPU threads at the same time next to each other, resulting in a higher degree of coalesced accesses [20].

VIII. Concluding remarks

We introduced *BigKernel*, a scheme that provides pseudo-virtual memory to Big Data-style GPGPU applications that operate on streaming data. BigKernel uses a 4-stage pipeline with automated prefetching to (i) optimize CPU-GPU communication and (ii) optimize GPU memory accesses. It simplifies the programming model by allowing programmers to write kernels using arbitrarily large data structures where the data records can be operated on independently, thus relieving the programmer from having to partition the data into segments, manage buffers, transfer data between CPU and GPU, and having to invoke GPU kernels multiple times. Straight-forward compiler transformations are used to transform traditional GPU kernels into BigKernel.

On six applications, we experimentally showed that BigKernel achieves an average speedup of 1.7X over implementations that use double buffering, and an average speedup of 3X over multi-core CPU implementations. We also showed that BigKernel largely removed PCIe from being a bottleneck for these applications, with the bottleneck migrating to the GPU cores.

As future work, we intend to gain more experience with additional applications with more complex data structures; in particular, we plan on applying BigKernel to MapReduce. Since GPU processing is now often the bottleneck for the applications we studied, we also intend to focus on improving GPU processing efficiency.

IX. Acknowledgments

The authors are grateful to the anonymous reviewers, Leonid Ryzhyk, and Ding Yuan for their constructive and helpful comments. This work is supported by NSERC research grants.

References

[1] M. Bauer, H. Cook, and B. Khailany. cudaDMA: Optimizing GPU Memory Bandwidth via Warp Specialization. In *Proc. 2011 Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 12:1–12:11, 2011.

[2] J. Chapman, I. Ho, S. Sunkara, S. Luo, G. Schroth, and D. Rokhsar. Meraculous: De Novo Genome Assembly with Short Paired-End Reads. *PLoS ONE*, (8):e23501, 2011.

[3] S. Chen and S. Schlosser. Map-reduce Meets Wider Varieties of Applications. *Intel Research Pittsburgh, Tech. Rep. IRP-TR-08-05*, 2008.

[4] R. Das, M. Uysal, J. Saltz, and Y. Hwang. Communication Optimizations for Irregular Scientific Computations on Distributed Memory Architectures. *Journal of Parallel and Distributed Computing*, pages 462–478, 1994.

[5] I. Gelado, J. Stone, J. Cabezas, S. Patel, N. Navarro, and W. Hwu. An Asymmetric Distributed Shared Memory Model for Heterogeneous Parallel Systems. In *Proc. 15th Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 347–358, 2010.

[6] C. Gregg and K. Hazelwood. Where is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer. In *Proc. IEEE Intl. Symp. on Performance Analysis of Systems and Software (ISPASS)*, pages 134–144, 2011.

[7] A. Hagiescu, H. Huynh, W. Wong, and R. Goh. Automated Architecture-Aware Mapping of Streaming Applications onto GPUs. In *Proc. 25th IEEE Intl. Parallel Distributed Processing Symp. (IPDPS)*, pages 467–478, 2011.

[8] T. Han and T. Abdelrahman. hiCUDA: a High-level Directive-based Language for GPU Programming. In *Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units (GPGPU)*, pages 52–61, 2009.

[9] H. Huynh, A. Hagiescu, W. Wong, and R. Goh. Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems. In *Proc. 17th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 1–10, 2012.

[10] T. Jablin, J. Jablin, P. Prabhu, F. Liu, and D. August. Dynamically managed data for CPU-GPU architectures. In *Proc. 10th Intl. Symp. on Code Generation and Optimization (CGO)*, pages 165–174, 2012.

[11] T. Jablin, P. Prabhu, J. Jablin, N. Johnson, S. Beard, and D. August. Automatic CPU-GPU Communication Management and Optimization. In *Proc. 32nd Conf. on Programming Language Design and Implementation (PLDI)*, pages 142–151, 2011.

[12] T. Komoda, S. Miwa, and H. Nakamura. Communication Library to Overlap Computation and Communication for OpenCL Application. In *Proc. 26th IEEE Intl. Parallel and Distributed Processing Symp. Workshops PhD Forum (IPDPSW)*, pages 567–573, 2012.

[13] S. Lee, S. Min, and R. Eigenmann. OpenMP to GPGPU: a Compiler Framework for Automatic Translation and Optimization. In *Proc. 14th Symp. on Principles and Practice of Parallel Programming (PPoPP)*, pages 101–110, 2009.

[14] S. Pai, R. Govindarajan, and M. Thazhuthaveetil. Fast and Efficient Automatic Memory Management for GPUs Using Compiler-assisted Runtime Coherence Scheme. In *Proc. 21st Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT)*, pages 33–42, 2012.

[15] S. Sharma, R. Ponnusamy, B. Moon, Y. Hwang, R. Das, and J. Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. In *Proc. of the 1994 Conf. on Supercomputing*, pages 97–106, 1994.

[16] S. Ueng, M. Lathara, S. Baghsorkhi, and W. Hwu. CUDA-Lite: Reducing GPU Programming Complexity. In *Languages and Compilers for Parallel Computing*, volume 5335, pages 1–15. 2008.

[17] T. Wilson, P. Hoffmann, S. Somasundaran, J. Kessler, J. Wiebe, Y. Choi, C. Cardie, E. Riloff, and S. Patwardhan. OpinionFinder: a System for Subjectivity Analysis. In *Proc. HLT/EMNLP on Interactive Demonstrations*, HLT-Demo '05, pages 34–35, 2005.

[18] Y. Yan, M. Grossman, and V. Sarkar. JCUDA: A Programmer-Friendly Interface for Accelerating Java Programs with CUDA. In *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 887–899. 2009.

[19] Y. Yang, P. Xiang, J. Kong, and H. Zhou. A GPGPU Compiler for Cemory Cptimization and Carallelism Canagement. In *Proc. 2010 Conf. on Programming Language Design and Implementation (PLDI)*, pages 86–97, 2010.

[20] E. Zhang, Y. Jiang, Z. Guo, K. Tian, and X. Shen. On-the-fly Elimination of Dynamic Irregularities for GPU Computing. In *Proc. 16th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 369–380, 2011.