Efficient Gather and Scatter Operations on Graphics Processors

Bingsheng He[#] Naga K. Govindaraju[*]
[#]Hong Kong Univ. of Science and Technology
{saven, luo}@cse.ust.hk

Qiong Luo[#] Burton Smith[*]
[*]Microsoft Corp.
{nagag, burtons}@microsoft.com

ABSTRACT

Gather and scatter are two fundamental data-parallel operations, where a large number of data items are read (gathered) from or are written (scattered) to given locations. In this paper, we study these two operations on graphics processing units (GPUs).

With superior computing power and high memory bandwidth, GPUs have become a commodity multiprocessor platform for general-purpose high-performance computing. However, due to the random access nature of gather and scatter, a naive implementation of the two operations suffers from a low utilization of the memory bandwidth and consequently a long, unhidden memory latency. Additionally, the architectural details of the GPUs, in particular, the memory hierarchy design, are unclear to the programmers. Therefore, we design multi-pass gather and scatter operations to improve their data access locality, and develop a performance model to help understand and optimize these two operations. We have evaluated our algorithms in sorting, hashing, and the sparse matrix-vector multiplication in comparison with their optimized CPU counterparts. Our results show that these optimizations yield 2-4X improvement on the GPU bandwidth utilization and 30-50% improvement on the response time. Overall, our optimized GPU implementations are 2-7X faster than their optimized CPU counterparts.

Categories and Subject Descriptors

E.5 [**Files**] Optimization, sorting/searching; I.3.1 [**COMPUTER GRAPHICS**] Hardware Architecture – *Graphics processors, parallel processing*.

General Terms

Algorithms, Measurement, Performance.

Keywords

Gather, scatter, parallel processing, graphics processors, cache optimization.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SC 2007, Nov. 10-16, Reno, NV, USA.
Copyright 2007 ACM 1595930612/07/0011 ...$5.00.

1. INTRODUCTION

The increasing demand for faster scientific routines to enable physics and multimedia applications on commodity PCs has transformed the graphics processing unit (GPU) into a massively parallel general-purpose co-processor [29]. These applications exhibit a large amount of data parallelism and map well to the data parallel architecture of the GPU. For instance, the current NVIDIA GPU has over 128 data parallel processors and a peak memory bandwidth of 86 GB/s. Many numerical algorithms including matrix multiplication [16][27][31], sorting [20][25] [30], LU decomposition [19], and fast Fourier transforms [24][28] have been designed on GPUs. Due to the high performance capabilities of GPUs, these scientific algorithms achieve 2-5X performance improvement over optimized CPU-based algorithms. In order to further improve the performance of GPU-based scientific algorithms and enable optimized implementations similar to those for the CPU-based algorithms, recent GPUs include support for inter-processor communication using shared local stores, and support for scatter and gather operations [6].

Scatter and gather operations are two fundamental operations in many scientific and enterprise computing applications. These operations are implemented as native collective operations in message passing interfaces (MPI) to define communication patterns across the processors [4], and in parallel programming languages such as ZPL [8] and HPF [1]. Scatter operations write data to arbitrary locations and gather operations read data from arbitrary locations. Both operations are highly memory intensive and form the basic primitives to implement many parallel algorithms such as quicksort [12], sparse matrix transpose [8], and others. In this paper, we study the performance of scatter and gather operations on GPUs.

Figure 1 shows the execution time of the scatter and the gather on a GPU with the same input array but either sequential or random read/write locations. The input array is 128MB. The detailed experimental setup is described in Section 3.5. This figure shows that location distribution greatly affects the performance of the scatter and the gather. For sequential locations, if we consider the data transfer of the output data array and internal data structures, both operations are able to achieve a system bandwidth of over 60 GB/s. In contrast, for random locations, both operations yield a low utilization of the memory bandwidth. The performance comparison indicates that locality in memory accesses is an important factor for the performance of gather and scatter operations on GPUs.

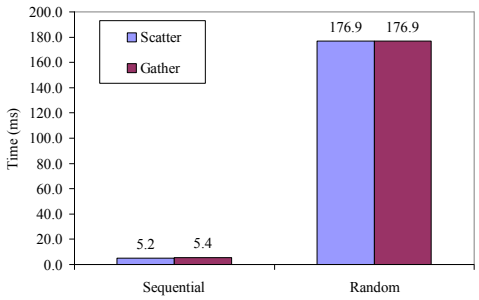

Figure 1. The elapsed time of the scatter and the gather on a GPU. Both operations underutilize the bandwidth for random locations.

GPU memory architectures are significantly different from CPU memory architectures [21]. Specifically, GPUs consist of high-bandwidth, high-latency video memory and the GPU cache sizes are significantly smaller than the CPUs – therefore, the performance characteristics of scatter and gather operations on GPUs may involve different optimizations than corresponding CPU-based algorithms. Additionally, it was only recently that the scatter functionality was introduced on GPUs, and there is little work in identifying the performance characteristics of the scatter on GPUs.

In this paper, we present a probabilistic analysis to estimate the performance of scatter and gather operations on GPUs. Our analysis accounts for the memory access locality and the parallelism in GPUs. Using our analysis, we design optimized algorithms for scatter and gather. In particular, we design multi-pass algorithms to improve the locality in the memory access and thereby improve the memory performance of these operations. Based on our analysis, our algorithms are able to determine the number of passes required to improve the performance of scatter and gather algorithms. Moreover, we demonstrate through experiments that our performance model is able to closely estimate the performance characteristics of the operations. As a result, our analysis can be applied to runtime systems to generate better execution plans for scatter and gather operations.

We use our scatter and gather to implement three common applications on an NVIDIA GeForce 8800 GPU (G80) - radix sort using scatter operations, and the hash search and the sparse-matrix vector multiplication using gather operations. Our results indicate that our optimizations can greatly improve the utilization of the memory bandwidth. Specifically, our optimized algorithms achieve a 2-4X performance improvement over single-pass GPU-based implementations. We also compared the performance of our algorithms with optimized CPU-based algorithms on high-end multi-core CPUs. In practice, our results indicate a 2-7X performance improvement over CPU-based algorithms.

Organization: The remainder of the paper is organized as follows. We give a brief overview of the GPU and the prior work on GPU- or CPU-based cache efficient algorithms in Section 2. Section 3 describes our performance model and presents our optimization techniques for GPUs. In Section 4, we use the optimized scatter and gather to improve the performance of sorting, the hash search and the sparse-matrix vector multiplication on the GPU. Finally, we conclude in Section 5.

2. BACKGROUND AND RELATED WORK

In this section, we first review techniques in general-purpose computing on the GPU. Next, we survey the cache-optimized techniques on the CPU and the GPU.

2.1 GPGPU (General-purpose computation on GPU)

A GPU is a SIMD processing unit with high memory bandwidth primarily intended for use in computer graphics rendering applications. In recent years, the GPU has become extremely flexible and programmable. This programmability is strengthened in every major generation (roughly every 18 months) [29]. Such programming flexibility further facilitates the use of the GPU as a general-purpose processor. Recently, NVIDIA has released the CUDA (Compute Unified Device Architecture) framework [6] together with the G80 GPU for general-purpose computation. Unlike DirectX or OpenGL, CUDA provides a programming model for a thread-based programming language similar to C. Thus, programmers can take advantage of GPUs without requiring graphics know-how. Two of the newly exposed features on the G80 GPU - data scattering and on-chip shared memory for the processing engines, are exploited by our implementation.

GPUs have been recently used for various applications such as matrix operations [16][27][31] and FFT computation [24][28]. We now briefly survey the techniques that use GPUs to improve the performance of scientific and database operations. The GPU-based scientific applications include linear algebra computations such as matrix multiplication [16] and sparse matrix computations [15]. Govindaraju et al. presented novel GPU-based algorithms for relational database operators including selections, aggregations [22] as well as sorting [20], and for data mining operations such as computing frequencies and quantiles for data streams [23]. For additional information on the state-of-the-art GPGPU techniques, we refer the reader to a recent survey by Owens et al. [29]. The existing work mainly develops OpenGL/DirectX programs to exploit the specialized hardware features of GPUs. In contrast, we use a general-purpose programming model to utilize the GPU hardware features, and implement our algorithms with two common building blocks, i.e., scatter and gather. Recently, Sengupta et al. implemented the segmented scan using the scatter on the GPU [31]. Our optimization on the scatter can be applied to the segmented scan to further improve its performance.

There is relatively less work on developing efficient algorithms for scatter and gather on GPUs, even though these two operations are commonly provided primitives in traditional MPI architectures [4]. Previous-generation GPUs support gather but do not directly support scatter. Buck described algorithms to implement the scatter using the gather [14]. However, these algorithms usually require complex mechanisms such as sorting, which can result in low bandwidth utilization. Even though new generation GPUs have more programming flexibility supporting both scatter and gather, the high programmability does not necessarily achieve high bandwidth utilization, as seen in Figure 1. These existing limitations motivate us to develop a performance model to understand the scatter and the gather on the GPU, and to improve their overall performance.

2.2 Memory optimizations

Memory stalls are an important factor for the overall performance of data-intensive applications, such as relational database systems [13]. For the design and analysis of memory-efficient algorithms, a number of memory models have been proposed, such as the external memory model [10] (also known as the cache-conscious model) and the cache-oblivious model [17]. Our model is similar to the external memory model but applies to the parallel computation on the graphics processor. In addition to modeling the cache cost, Bailey proposed a memory model to estimate the cost of memory bank contention on the vector computer [11]. In contrast, we focus on the cache performance of the graphics processor.

The algorithms reducing the memory stalls can be categorized into two kinds, cache-conscious [32] and cache-oblivious [17]. Cache-conscious algorithms utilize knowledge of cache parameters, such as cache size. On the other hand, cache-oblivious algorithms do not assume any knowledge of cache parameters. Cache-conscious techniques have been extensively used to improve the memory performance of the CPU-based algorithms. LaMarca et al. [26] studied the cache performance for the quick sort and showed that cache optimizations can significantly improve the performance of the quick sort. Boncz et al. [13] proposed the radix clustering with a multi-pass partitioning method in order to optimize the cache performance. Williams et al. [33] proposed cache optimizations for matrix operations on the Cell processor.

Memory optimization techniques have also been shown useful for GPU-based algorithms. These memory optimizations need to adapt to the massive threading architecture of the GPU. Govindaraju et al. [21] proposed a memory model for scientific applications and showed that optimizations based on the model greatly improved the overall performance. Fatahalian et al. [16] analyzed the cache and bandwidth utilization of the matrix-matrix multiplication. Galoppo et al. [19] designed block-based data access patterns to optimize the cache performance of dense linear systems. In comparison, we propose a general performance model for scatter and gather, and use a multi-pass scheme to improve the cache locality.

3. MODEL AND ALGORITHM

In this section, we first present our memory model on the graphics processor. Next, we give the definitions for the scatter and the gather. We then present our modeling and optimization techniques for the scatter and the gather. Finally, we present our evaluation results for our techniques.

3.1 Memory model on GPU

Current GPUs achieve a high memory bandwidth using a wide memory interface, e.g., the memory interface of G80 is 384-bit. In order to mask high memory latency, GPUs have small L1 and L2 SRAM caches. Compared with the CPU, the GPU typically has a flat memory hierarchy and the details of the GPU memory hierarchy are often missing in vendor hardware specifications.

We model the GPU memory as a two-level hierarchy, the cache and the device memory, as shown in Figure 2. The data access to the cache can be either a hit or a miss. If the data is found in the cache, this access is a hit. Otherwise, it is a miss and a memory block is fetched from the device memory. Given the number of cache misses, *#miss*, and that of cache hits, *#hit*, the cache hit rate, h, is defined as $h=\frac{\#hit}{\#hit+\#miss}$.

We model the GPU as M SIMD multiprocessors. At any time, all processors in a multiprocessor execute the same instruction. The threads on each multiprocessor are grouped into thread warps. A *warp* is the minimum schedule unit on a multiprocessor. If a warp is stalled by memory access, it will be put to the end of a waiting list of thread warps. Then, the first warp in the waiting list becomes active and is scheduled to run on the multiprocessor.

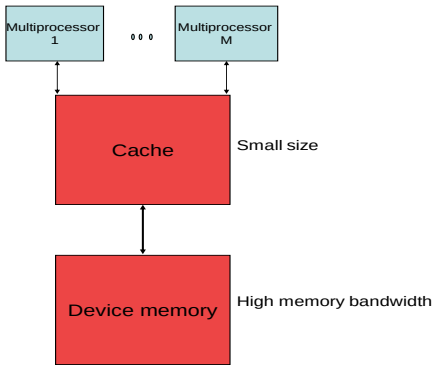

Figure 2. The memory model of GPU. The GPU consists of *M* SIMD multiprocessors. The cache is shared by all the multiprocessors.

3.2 Definitions

Gather and scatter are dual operations. A scatter performs indexed writes to an array, and a gather performs indexed reads from an array. We define the two operations in Figure 3. The array L for the scatter contains *distinct* write locations for each R_{in} tuple, and that for the gather the read locations for each R_{out} tuple. Essentially, the scatter consists of sequential reads and random writes. In contrast, the gather consists of sequential writes and random reads. Figure 4 illustrates examples for the scatter and the gather.

Primitive: **Scatter** Input: R_{in} [1, …, n], L[1, …, n]. Output: R_{out} [1, …, n]. Function: $R_{out}[L[i]]=R_{in}[i]$, i=1, …n.	Primitive: **Gather** Input: R_{in} [1, …, n], L[1, …, n]. Output: R_{out} [1, …, n]. Function: $R_{out}[i]=R_{in}[L[i]]$, i=1, …n.

Figure 3. Definitions of scatter and gather

To quantify the performance of the scatter and gather operations, we define the *effective bandwidth* for the sequential access (resp. the random access) to be the ratio of the amount of the data array tuples accessed by the sequential access (resp. the random access) in the scatter and gather operations to the elapsed time. This measure indicates how much bandwidth is utilized to access the target (effective) data. That means, we define two kinds of effective bandwidth, *sequential* and *random bandwidths*, to distinguish the sequential and the random access patterns, respectively. Since the sequential access has a better locality than the random access, the sequential bandwidth is usually higher than the random bandwidth.

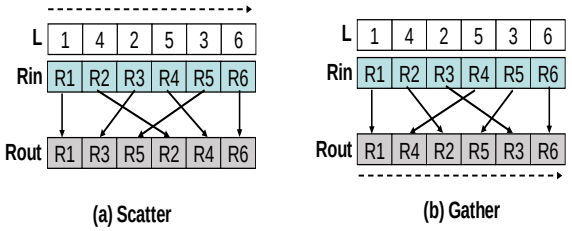

Figure 4. Examples of scatter and gather.

3.3 Modeling the effective bandwidth

In this subsection, we present our probabilistic model on estimating the sequential bandwidth and the random bandwidth (denoted as B_{seq} and B_{rand}). Since the estimation and optimization techniques are similar on the scatter and the gather operations, we describe the techniques for the scatter in detail, and present those for the gather in brief.

3.3.1 Modeling the sequential bandwidth

A sequential access fetches each memory block exactly once. A memory block causes a cache miss at the first access, and the later accesses to this memory block are cache hits. Thus, the sequential access achieves a high effective bandwidth.

We estimate the sequential bandwidth using our measured effective bandwidth. To obtain the measured bandwidth, we perform a set of calibration measurements with input data arrays of different sizes. Given the capacity of the device memory, D_m MB, we measure the execution time for the sequential scan when the input array size is 2^i MB, i=1, 2, …, $^{log_2 D_m}$. Let the execution time for the input array of 2^i MB be t_i. We compute the sequential bandwidth to be the weighted bandwidth of all calibrations as Eq. 1, where n= $^{log_2 D_m}$. Compared with the estimation using the average value, the weighted sum gives more weight to the bandwidth of the larger sizes, which fully utilize the bus bandwidth of the GPU.

$$B_{seq} = \frac{1}{\sum_{i=1}^{n} 2^i} \cdot \sum_{i=1}^{n} \left(\frac{2^i}{t_i} \cdot 2^i\right) \quad (1)$$

3.3.2 Modeling the random bandwidth

Compared with the sequential access, the random access has a low cache locality. Some accesses are cache hits, and many others are misses. Given the cache hit rate, h, the expected time for accessing a data array is defined as t, given in Eq. 2.

$$t = n \cdot ((1-h) \cdot \frac{\lceil z/l \rceil \cdot l}{B_{seq}} + h \cdot \lceil z/l' \rceil \cdot \delta) \quad (2)$$

In this equation, n is the number of data array tuples; z is the size of each tuple (bytes); l and l' are the memory block size and the transfer unit between the cache and the GPU (bytes); B_{seq} is the estimated sequential bandwidth; δ is the time for fetching a cache line from the cache to the GPU. In this equation, the amortized cost for a cache hit is $\lceil z/l' \rceil \cdot \delta$. The data transfer between the device memory and the GPU for a cache miss is $\lceil z/l \rceil \cdot l$. Since B_{seq} measures the data transfer between the device memory and the GPU as well as the accesses to the GPU cache, we estimate the total cost for a cache miss to be $\frac{\lceil z/l \rceil \cdot l}{B_{seq}}$.

We define the random bandwidth as follows.

$$B_{rand} = \frac{n \cdot z}{t} = \frac{z}{\frac{\lceil z/l \rceil \cdot l}{B_{seq}} + h \cdot (\lceil z/l' \rceil \cdot \delta - \frac{\lceil z/l \rceil \cdot l}{B_{seq}})} \quad (3)$$

Given a fixed input, we maximize the cache hit rate in order to maximize the random bandwidth.

We develop a probabilistic model to estimate the cache hit rate. Specifically, we estimate the cache hit rate in two cases. One is that the access locations are totally random, and the other is with the knowledge of *the number of partitions* in the input data. The number of partitions is usually a priori knowledge for various applications such as radix sort and hash partitioning. For instance, given the number of bits used in the radix sort, *Bits*, the number of partitions for the input data is 2^{Bits}. With the knowledge of the number of partitions, we have a more accurate estimation on the cache hit rate. When there is no knowledge of the number of partitions, we assume the access locations are random and use the first case to estimate the cache hit rate.

Case I: totally random accesses. We first estimate the expected number of distinct memory blocks, E, accessed by n random accesses. Suppose the total number of memory blocks for storing the input data array is k. Let E_j be the number of cases accessing exactly j ($1<=j<=n$) distinct memory blocks. We estimate E_j as follows.

$$E_j = C(k, j) \cdot S(n, j) \cdot j! \quad (4)$$

In this equation, $C(k, j)$ is the number of cases of choosing j memory blocks from the available k memory blocks; $S(n,j)$ is the Stirling number of the second kind [9] that equals the number of cases of partitioning n accesses into j memory blocks; $j!$ is the number of cases in the permutation of the j memory blocks. Since the total number of cases in accessing the k memory blocks is k^n, we estimate the expected number of distinct memory blocks by these n accesses in Eq. 5.

$$E = \frac{\sum_{j=1}^{n} (E_j \cdot j)}{k^n} \quad (5)$$

Based on E and the number of cache lines in the GPU cache, N, we estimate the cache miss rate, m, in Eq. 6.

$$m = \begin{cases} \frac{E}{n} & , E \le N \\ \frac{E + (1 - \frac{N}{E}) \cdot (n - E)}{n} & , otherwise \end{cases} \quad (6)$$

When $E>N$, we model cache misses in two categories -- the compulsory misses of loading E memory blocks, and the capacity misses, $^{(1-\frac{N}{E})(n-E)}$. Intuitively, given a fixed input, the larger the cache size, the smaller the cache miss rate.

Finally, we compute the cache hit rate of Case I to be, *h=(1-m)*.

Case II: Given the number of partitions, *p*. In this case, if the write locations of a thread warp belong to the same partition, they are consecutive. The writes to consecutive locations are sequential, and have a good cache locality.

Suppose a warp consists of *w* threads. Each thread accesses one tuple at one time. The number of tuples accessed by the warp is *w*. To estimate the number of cache misses, we estimate the expected number of distinct partitions, *D*, accessed by the warp. Let *Dj* be the number of cases containing exactly *j* distinct partitions, 1<=*j*<=*min(w, p)*. We estimate *Dj* to be *Dj=C(p,j)*S(w,j)*j!*. The estimation is similar to that on the number of distinct memory blocks accessed in Case I. Since the total number of cases is p^w, we estimate *D* in Eq. 7.

$$D = \frac{\sum_{j=1}^{\min(p,w)} (D_j \cdot j)}{p^w} \quad (7)$$

The average number of tuples belonging to each of these *D* partitions is $\frac{w}{D}$. The number of memory blocks for each partition is $\left\lceil \frac{w \cdot z}{D \cdot l} \right\rceil$. Therefore, the cache miss rate within a thread warp is estimated in Eq. 8.

$$m = \frac{D \cdot \left\lceil \frac{w \cdot z}{D \cdot l} \right\rceil}{w} \quad (8)$$

The cache hit rate within a thread warp to be *(1-m)*.

Finally, we compute the cache hit rate of Case II to be the sum of the intra- and inter-warp cache hit rates. The inter-warp cache hit rate is estimated as in Case I, totally random accesses. Note, even with the knowledge of the number of partitions, the accesses of warps on different multiprocessors are random due to the unknown execution order among warps.

3.4 Improving the memory performance

A basic implementation of the scatter is to sequentially scan *L* and R_{in} once, and output all R_{in} tuples to R_{out} during the scan. Similarly, the basic implementation of the gather is to scan *L* once, read the R_{in} tuples according to *L*, and write the tuples to R_{out} sequentially. Given B_{seq} and B_{rand}, the total execution time of the scatter and the gather is estimated in Eq. 9 and 10, respectively. Given an array, *X*, we denote the number of tuples in *X* and the size of *X* (bytes) to be |*X*| and ||*X*||, respectively.

$$T_{scatter} = \frac{\| R_{in} \|}{B_{seq}} + \frac{\| L \|}{B_{seq}} + \frac{\| R_{out} \|}{B_{rand}} \quad (9)$$

$$T_{gather} = \frac{\| R_{in} \|}{B_{rand}} + \frac{\| L \|}{B_{seq}} + \frac{\| R_{out} \|}{B_{seq}} \quad (10)$$

This basic implementation is simple. However, if *L* is random, the scatter and the gather suffer from the random access, which has low cache locality and results in a low bandwidth utilization.

Since B_{rand} is usually much lower than B_{seq}, we consider a multi-pass optimization scheme to improve the cache locality. We apply the optimization technique to both the scatter and gather operations, and illustrate the optimization technique using the scatter as an example.

Suppose we perform the scatter in *nChunk* passes. In each pass, we output only those R_{in} tuples that belong to a certain region of R_{out}. That is, the algorithm first divides R_{out} into *nChunk* chunks, and then performs the scatter in *nChunk* passes. In the *i*th pass (1<=*i*<=*nChunk*), it scans *L* once, and outputs the R_{in} tuples belonging to the *i*th chuck of R_{out} (i.e., the write locations are between $(i-1) \cdot \frac{|R_{out}|}{nChunk}$ and $i \cdot \frac{|R_{out}|}{nChunk}$). Since each chunk is much smaller than $|R_{out}|$, our multi-pass scatter has a better cache locality than the single-pass one.

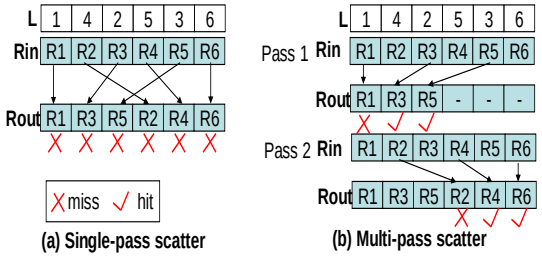

Figure 5. Single-pass vs. multi-pass scatter: the multi-pass scatter has a better cache locality than the single-pass one.

Let us illustrate our multi-pass scheme using an example, as shown in Figure 5. For simplicity, we assume that the GPU cache can hold only one memory block and a block can hold three tuples. Under this assumption, each write in the single-pass scheme in Figure 5 (a) results in a miss. Consequently, the single-pass scheme has a low cache locality. In comparison, the multi-pass scheme divides R_{out} into two chunks, and writes one chunk in each pass. It achieves an average hit rate of 2/3.

We estimate the total execution time of the multi-pass scatter as scanning R_{in} and *L* for *nChunk* times and outputting the data. Given the *nChunk* value, we estimate the cache hit rate in a similar way to the single-pass scheme. The major difference is that in the estimation of the random bandwidth, the number of data accesses considered in each pass is *n/nChunk*, where *n* is the total number of data accesses in the scatter. When *nChunks>1*, *n/nChunks* is smaller than *n*, the multi-pass scheme has a higher cache hit rate than the single-pass scheme.

Given the estimated random bandwidth for the multi-pass scheme when the number of passes is *nChunk*, B_{rand}, we estimate the execution time of the multi-pass scatter, $T'_{scatter}$, in Eq. 11. We choose the *nChunk* value so that $T'_{scatter}$ is minimized.

$$T'_{scatter} = \frac{(\| R_{in} \| + \| L \|) \cdot nChunk}{B_{seq}} + \frac{\| R_{out} \|}{B_{rand}} \quad (11)$$

3.5 Evaluation

We evaluated our model and optimization techniques with different data sizes and data distributions. We increase the data size to be sufficiently large to show the performance trend. The default tuple size is 8 bytes, and the default size of the input data array is 128MB (i.e., the input data array consists of 16 million tuples). The default data distribution is random.

We have implemented and tested our algorithms on a PC with an NVidia G80 GPU. The PC runs Windows XP on an Intel Core2

Quad CPU. We also run our CPU-based algorithms on a Windows Server 2003 server machine with two AMD Opteron 280 dual-core processors. The hardware configuration of the PC and the server is shown in Table 1. The cache configuration of the GPU is obtained based on the GPU memory model proposed by Govindaraju et al. [21]. The cache latency information of the CPU are obtained using a cache calibrator [1], and those of the GPU are obtained from the hardware specification [6]. We estimate the transfer unit between the cache and the GPU to be the memory interface, i.e., 384 bits, since we do not have any details on this value.

We compute the theoretical memory bandwidth to be the bus width multiplying the memory clock rate. Thus, the GPU, Intel and AMD have a theoretical bandwidth of 86.4 GB/s, 10.4 GB/s and 8.0 GB/sec, respectively. Based on our measurements on sequentially scanning a large array, the G80 achieves a peak memory bandwidth of around 69.2 GB/s whereas the Intel and the AMD 5.6 GB/s and 5.3 GB/s, respectively.

To evaluate our model, we define the accuracy of our model to be $(1-\frac{|v-v'|}{v})$, where v and v' are the measured and estimated values, respectively. We run each experiment five times and report the average value. Since we aim at validating our cache model and evaluating the scatter and the gather on the GPU, the results for the GPU-based algorithms in this section do not include the data transfer time between the CPU and the GPU.

Table 1. Hardware configuration

	GPU(G80)	*CPU(Intel)*	*CPU(AMD)*
Processors	1350MHz × 8 × 16	2.4 GHz × 4 (Quad-core)	2.4GHz × 2 × 2 (two dual-core)
Cache size	392 KB	L1: 32KB × 4, L2: 8MB	L1: 64KB × 2 × 2, L2: 1MB × 2 × 2
Cache block size (bytes)	256	L1: 64, L2: 128	L1: 128, L2: 128
Cache access time (cycle)	10	L1: 3, L2: 11	L1: 3, L2: 9
DRAM (MB)	768	1024	16384
DRAM latency (cycle)	200	138	138
Bus width (bit)	384	64	64
Memory clock (GHz)	1.8	1.3	1.0

Validating the performance model. Figure 6 demonstrates the measured and estimated performance of sequential scatter and gather operations on the GPU. The gather and the scatter have a very similar performance trend as the data size increases. The sequential bandwidth is estimated to be 63.9 GB/s. The figure also indicates that our estimation achieves an accuracy of over 87% for the gather and the scatter. The range of the accuracy is 80%~97%, i.e., min=70%, max=99%.

Figure 7 shows the measured and estimated performance with the data size varied when the read/write locations are totally random. Our model has a high accuracy on predicting the performance of the gather and the scatter. The average accuracy of our model on the gather and the scatter is over 90% (min: 62%, max: 98%). Again, the gather and the scatter have a very similar performance trend as the data size increases. In the following, we report the results for the scatter only, because the results of the gather are similar to those of the gather.

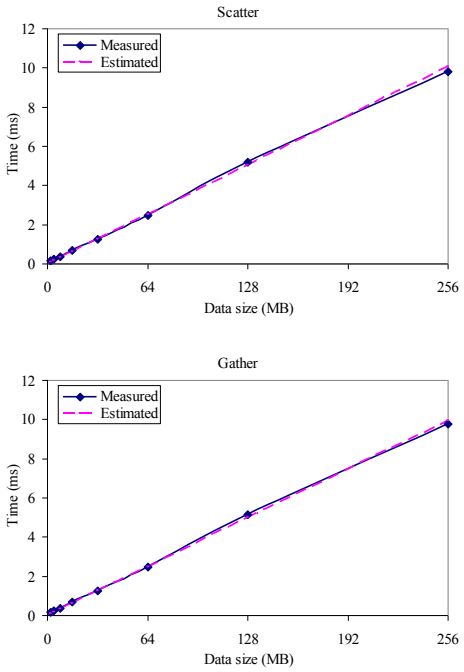

Figure 6. The measured and estimated performance of the sequential single-pass scatter (top) and gather (bottom) on the GPU.

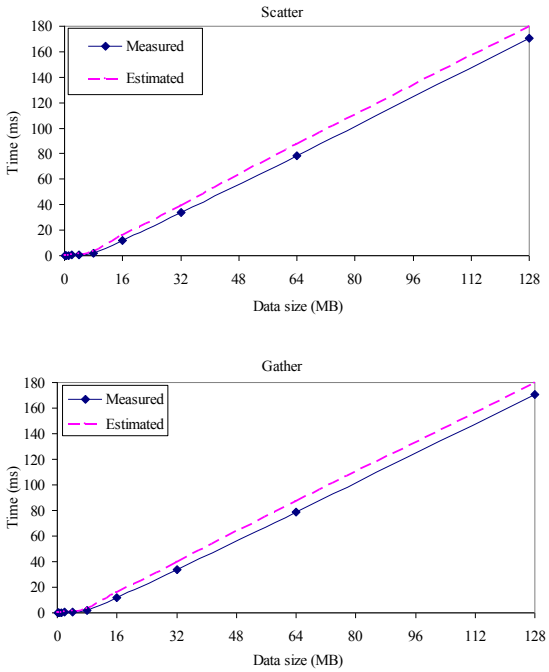

Figure 7. The measured and estimated performance of the single-pass scatter (top) and gather (bottom) on the GPU with random locations.

Figure 8 shows the measured and estimated performance with the number of partitions in the input data varied. The data size is

128MB. Given the number of partitions, the partition ID of each tuple is randomly assigned. When the number of partitions is larger than 32 (i.e., the warp size on the G80), the measured time increases dramatically due to the reduced cache reuse within a warp. We also observe this performance increase in our estimated time. These figures indicate that our estimation is accurate on different data sizes and different numbers of partitions in the input data. The average accuracy in Figure 8 is over 82% (min: 70%, max: 99%).

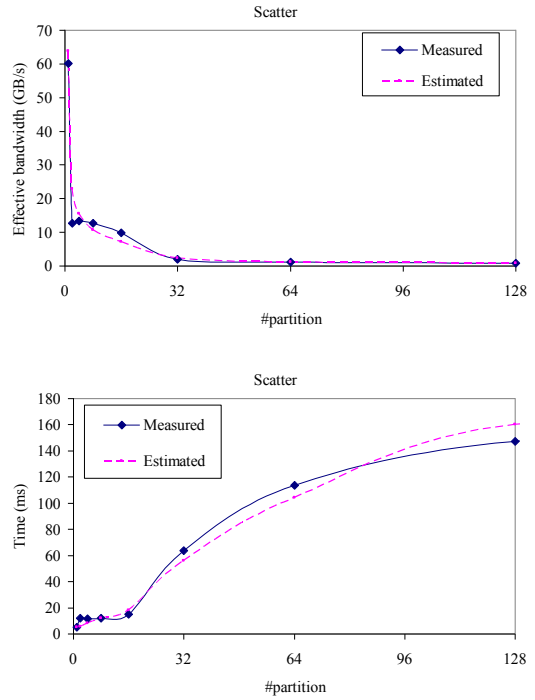

Figure 8. The measured and estimated single-pass scatter performance on the GPU: (top) the effective bandwidth of the random access; (bottom) the execution time.

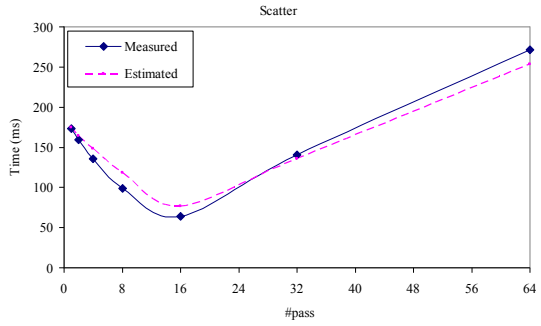

Figure 9. The measured and estimated performance of the multi-pass scatter on GPU with the number of passes varied.

Finally, we validate our performance model on the multi-pass scheme. Figure 9 shows the measured and estimated performance of the multi-pass scatter with the number of passes varied. The suitable value for the number of passes is 16. When the number of passes is smaller than the suitable value, the scatter performance improves because the cache locality of the scatter is improved. When the number of passes is larger than the suitable value, the overhead of extra scans in the multi-pass scheme becomes significant. The average accuracy of our model in Figure 9 is 86% (min: 70%, max: 99%), and our estimations for different data sizes and different number of partitions are also highly accurate.

Evaluating the multi-pass scheme. We first consider whether we can improve the performance through dividing the input array into multiple segments. We apply our multi-pass scheme to each segment. Since the segment can fit into the cache, the cost of the multiple scans is minimized. Figure 10 shows the scatter performance when the number of segments, #bin, increases. The input array is 128 MB. When #bin is 512, the segment size is 256KB, which is smaller than the GPU cache size. When the #bin value increases, the scatter time increases as well. This is because the cache locality between the scatters on different segments gets worse.

We next evaluate our multi-pass scheme with the number of partitions, *p*, varied. The result is shown in Figure 11. When $p \le 8$, the suitable number of passes in our multi-pass scheme is one, and the multi-pass scatter reduces to the single-pass one. As *p* increases from 8, our multi-pass scheme outperforms the single-pass scheme. Regardless of the number of partitions in the input relation, our model correctly predicts a suitable value for the number of passes, and our multi-pass optimization technique improves the scatter performance up to three times.

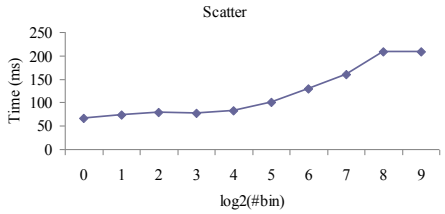

Figure 10. The scatter performance with the number of segments varied.

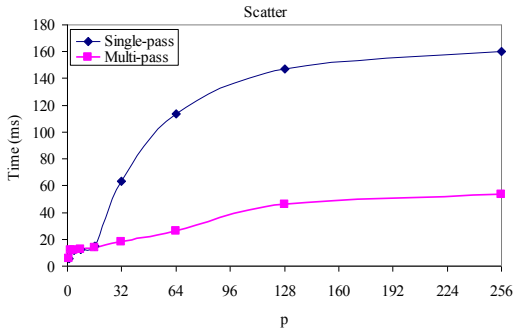

Figure 11. The performance of the multi-pass scatter on the GPU with the number of partitions, *p*, varied.

Finally, we compare the performance of scatter on the GPU and the CPU when the write locations are random. We obtain similar performance results as the number of partitions varied, and omit the results. Figure 12 shows the scatter performance with the tuple size varied and with the number of tuples fixed to be 16 million. The maximum tuple size is 16 due to the limited size of the device memory. Figure 13 shows the elapsed time with the number of tuples varied and the tuple size fixed to be 8 bytes.

On the CPU, we consider whether our multi-pass scheme and the multithreading technique can improve the performance of the scatter. We find that the multi-pass scheme has little performance improvement on both Intel and AMD. This is due to the difference of memory architectures of the GPU and the CPU. We apply the single-pass scheme to the gather and the scatter on the CPU. The multithreading technique improves the performance of the scatter by 1-2X and 2-4X on Intel and AMD, respectively.

On the GPU, the effective bandwidth of the random access increases as the tuple size increases. When the tuple size is fixed, the effective bandwidth is stable as the data size increases. Regardless of the tuple size and the data size, the GPU-based scatter has a much better performance than the optimized CPU-based one. The speedup is 7-13X and 2-4X on Intel and AMD, respectively. On both the GPU and CPUs, the effective bandwidth is much lower than the peak bandwidth. This indicates the random access pattern has low locality and yield low bus utilization. Nevertheless, our multi-pass scheme improves the effective bandwidth of the GPU-based scatter by 2-4X.

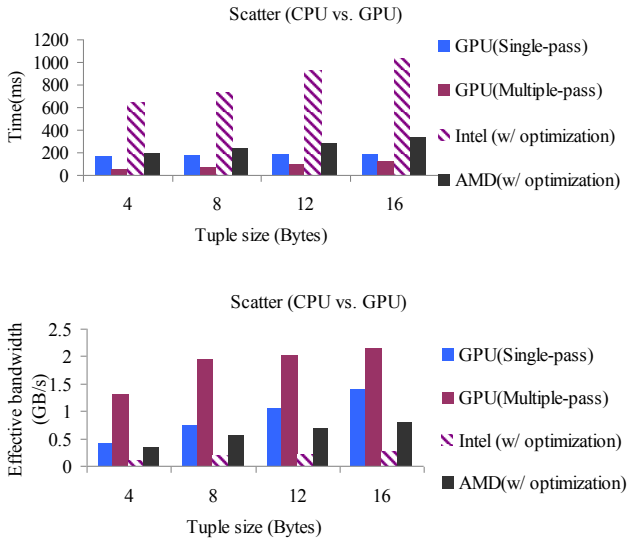

Figure 12. The scatter performance on the CPU and the GPU with tuple size varied.

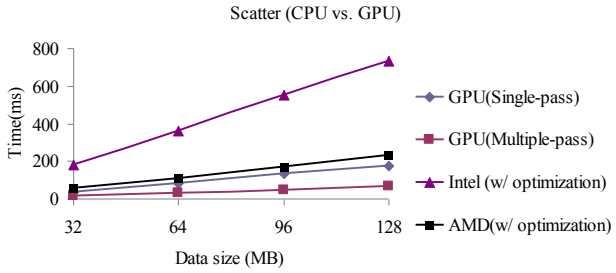

Figure 13. The scatter time on the CPU and the GPU with data size varied.

4. APPLICATION AND ANALYSIS

Scatter and gather are two fundamental data-parallel operations for many parallel algorithms [29]. In this section, we use scatter and gather to implement three memory intensive algorithms on GPUs, including the radix sort, the hash search and the sparse-matrix vector multiplication. Specifically, we use our improved scatter to implement the radix sort and our improved gather for the hash search and the sparse-matrix vector multiplication. We also compare the end-to-end performance of the three applications on the GPU and the CPU. Our total execution time for the GPU-based algorithms includes the data transfer time between the CPU and the GPU.

4.1 Radix sort

Radix sort is one of the highly parallel sorting algorithms [34]. It has a linear-time computational complexity. In this study, we implement the most significant digit (MSD) radix sort. The algorithm first performs *P*-phase partitioning on the input data, and then sorts the partitions in parallel. In each phase, we divide the input data into multiple partitions using *Bits* bits. In the *i*th ($0<=i<=P$) phase of the radix sort, we use the (i**Bits*)th to ((i+1)**Bits*)th bits from the most significant bit. After partitioning, we use the bitonic sort [20] to sort each partition. The bitonic sort can exploit the fast shared memory on G80 [6], where the shared memory of each multiprocessor is 16KB. Different from the cache, the shared memory is programmer-manipulated. Additionally, it is shared by threads on the same multiprocessor. When the partition size is smaller than the shared memory size, the bitonic sort works entirely on the data in the shared memory, and causes no further cache misses. We determine the suitable *P* value so that the resulting size of each partition is smaller than the shared memory size, i.e., $P=log_f(n/S)$, where f is the number of partitions generated in each phase, i.e., 2^{Bits}; n, the size of the input array (bytes); S, the shared memory size of each multiprocessor (bytes).

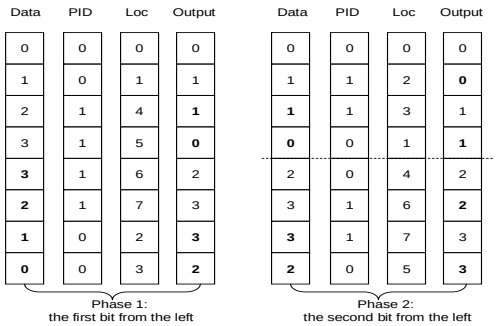

Figure 14. An example of radix sort.

We implement the partitioning scheme using the scatter. Given an input array of data tuples, *dataArray*, the partitioning process is performed in three steps. The algorithm first computes the partition ID for each tuple and stores all partition IDs into an array, *pidArray*, in the same order as *dataArray*. Next, it computes the write location for each tuple based on *pidArray*, and stores the write locations into the third array, *locArray*. Finally, it applies our multi-pass scatter to output *dataArray* according to *locArray*. An example of the radix sort on eight tuples is illustrated in Figure 14. The first step is fast, because it is a sequential scatter and a scan. We consider optimizations on the second and third steps.

For the second step, we use histograms to compute the write location for each tuple. Our histograms are similar to those in the

parallel radix sort proposed by Zagha [34]. The main difference is that we fully consider the hardware features of the GPU to design our histograms. Specifically, in our partitioning algorithm, each thread is responsible for a portion of the input array. Because the histogram for each thread is accessed randomly, we store the histograms into the shared memory to reduce the latency of these random accesses.

In our implementation, each histogram element is encoded in one byte. The size of each histogram is 2^{Bits} bytes. Since each thread has one histogram, the *Bits* value must satisfy the following constraint due to the limited size of the shared memory: $T*2^{Bits}<=S$, where T is the number of concurrent threads sharing the shared memory and S is the size of the shared memory.

For the third step, we determine the suitable number of passes to minimize the execution time of the radix sort. The number of phases, i.e., the number of scatter operations, in the radix sort is P. We estimate the execution time of each scatter as in Case II with the number of partitions, 2^{Bits}. The total execution time of these P scatter operations is $C=P*T'_{scatter}$. Considering the constraint in the second step, we choose the suitable *Bits* value to minimize the C value.

We have implemented the radix sort on the GPU and the CPU. Figure 15 shows the performance impact of our optimization techniques on the GPU and the CPU. Each array element is an 8-byte record, including a four-byte key value and a four-byte record identifier. The key values are randomly distributed.

For the CPU implementation, we consider both multithreading and cache-optimization techniques. Suppose the CPU has C_p cores. We first divide the input array into C_p partitions, and sort these partitions in parallel. Each partition is sorted by a thread running an optimized CPU-based radix sort, which is a multi-pass algorithm [13]. It determines the number of bits used in each pass according to the cache parameters of the CPU, e.g., the number of cache lines in the L1 and the L2 caches and the number of entries in the data TLB (Translation Lookaside Buffer). According to the existing cost model [13], the number of bits in each pass is set to be six on both Intel and AMD so that the cache stalls are minimized. The number of partitions generated in one pass is 2^6=64. Finally, we merge the C_p sorted partitions into one. The performance comparison for the CPU-based radix sort with and without optimizations is shown on the top of Figure 15**.** Our optimization techniques greatly improve the radix sort by 85% on AMD and slightly improve the radix sort on Intel.

On the GPU, the number of bits used in each phase of the radix sort, *Bits*, is five. That is, in each phase, an array is divided into 32 sub-arrays. Our test data takes three phases of partitioning in total. The suitable number of passes in our optimized scatter is four according to our performance model. As shown in Figure 15, our multi-pass scheme improves the GPU-based radix sort by over 30%. Note, the data transfer time between the GPU and the CPU is 10-30% of the total execution time of the GPU-based radix sort.

Figure 16 highlights the performance of the GPU-based radix sort, the GPU-based bitonic sort [20] and the CPU-based radix sort [13]. The GPU-based radix sort is twice faster than the GPU-based bitonic sort. Moreover, our GPU-based algorithm is over 2X faster than the best optimized CPU-based radix sort.

Finally, we perform back-of-envelope calculation on the system bandwidth. We estimate the data transfer of the GPU-based radix sort through considering the cache misses estimated by our model and the internal data structures in the radix sort. The CPU-based radix sort is estimated using the existing cost model [13]. The estimated bandwidth of the radix sort is on average 21.4 GB/s, 1.2 GB/s, and 2.1 GB/s on the GPU, Intel and AMD, respectively.

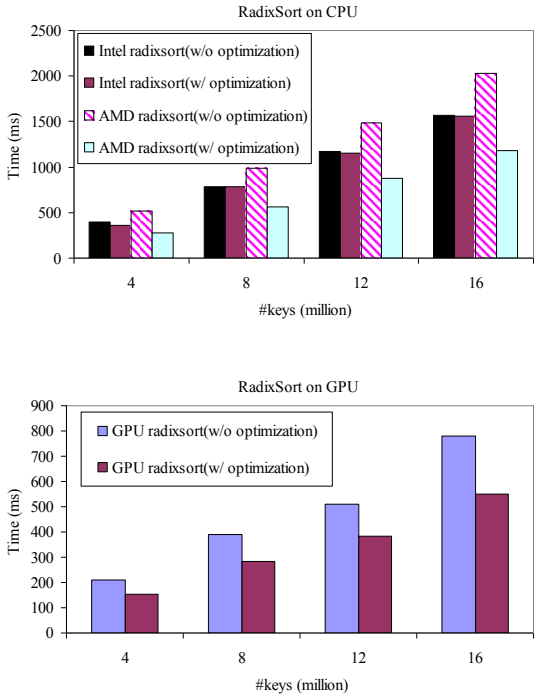

Figure 15. The performance impact of optimizations on the CPU and the GPU.

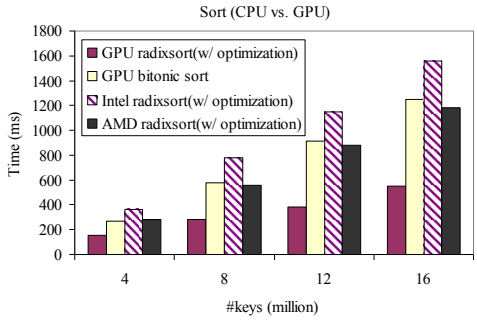

Figure 16. The execution time of sorting algorithms on the GPU and the CPU.

4.2 Hash search

A hash table is commonly used to speed up equality search on a set of records. Applications using hash tables include the hash join and hash partitioning in databases [13]. A hash table consists of multiple hash headers, each of which maintains a pointer to its corresponding bucket. Each bucket stores the key values of the records that have the same hash value, and the pointers to the records. An example of the hash table is illustrated in Figure 17

(a). Since dynamic allocation and deallocation of the device memory is not supported on current GPUs, we use an array to store the buckets, and each header maintains the start position of its corresponding bucket in the array.

The hash search takes as input a number of search keys, performs probes on a hash table with these keys, and outputs the matching records to a result buffer in the device memory. In our implementation, each thread is responsible for one search key. To avoid the conflict when multiple threads are writing results to the shared result buffer concurrently, we implement the hash search in four steps. First, for each search key, we use the hash function to compute its corresponding bucket ID and determine the (start, end) pair indicating the start and end locations for the bucket. Second, we scan the bucket and determine the number of matching results. Third, based on the number of results for each key, we compute a prefix sum on these numbers to determine the start location at which the results of each search key are written, and output the record IDs of the matching tuples to an array, *L*. Fourth, we perform a gather according to *L* to fetch the actual result records. Due to the random nature of the hash search, we estimate the execution time of the gather in Case I. Figure 17 (b) illustrates the four steps of searching two keys.

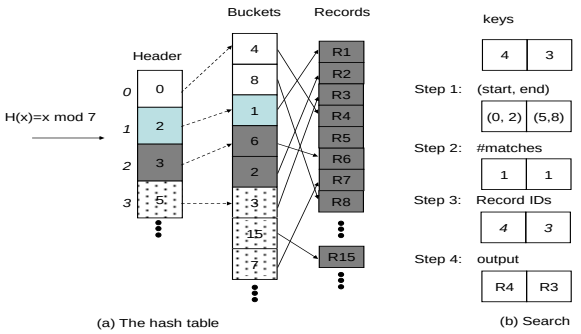

Figure 17. An example of hash search.

Figure 18 shows the execution time of the hash search on the GPU and the CPU. Note that the data transfer time is 40-60% of the total execution time of the hash search. The values of the search keys and the keys in the hash table are all 32-bit random integers. The record size is 8 bytes. We fixed the number of search keys to be 4 million and varied the total number of records in the hash table. The load factor of the hash table is around 99%. Similar to the CPU-based radix sort, we consider multithreading and cache optimization techniques for the CPU-based hash search. Suppose the CPU has C_p cores. We first divide the search keys into C_p partitions, and probe the hash table using the search keys of each partition in parallel. In our implementation, we used C_p threads and each thread is in charge of one partition. Each thread runs the cache-efficient hash searches using a multi-pass scheme similar to that on the GPU. The difference is that the chunk size is set to be the L2 data cache size. Similar to the CPU-based radix sort, the performance improvement of our optimization on AMD is significant and that on Intel is moderate.

Based on our performance model, the suitable number of passes for the GPU-based multi-pass scatter is 16. We observe that the optimized GPU-based hash search is 7.2X and 3.5X faster than its optimized CPU-based counterpart on Intel and AMD, respectively. Additionally, the hash search optimized with our multi-pass scheme is on average 50% faster than the single-pass scheme. We also varied the number of search keys and the number of records in the hash table, and obtained similar performance results. Through back-of-envelope calculation, the estimated bandwidth of the hash search is on average 13.0 GB/s, 0.56 GB/s, and 1.0 GB/s on the GPU, Intel and AMD, respectively.

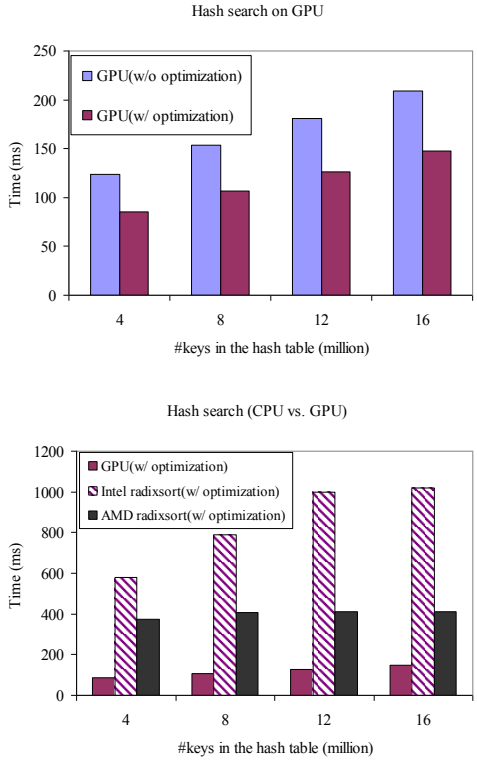

Figure 18. Hash search on the GPU and the CPU.

4.3 Sparse-matrix vector multiplication (SpMV)

Sparse-matrix vector multiplication (SpMV) is widely used in many linear solvers [15]. SpMV stores the sparse matrix in a compressed format and performs the multiplication on the compressed matrix. We adopt the compressed format (CSR) from the Intel MKL library [3] to represent the sparse matrix: for each row, only the non-zero elements are stored. In row *i*, if column *j* is not zero, we represent this cell as a tuple (*j*, *v*), where *v* is the value of the cell (*i*, *j*). This compressed format stores the matrix into four arrays: *values*, *columns*, *pointerB* and *pointerE*. Array *values* stores all the values of non-zero cells in the order of their row and column indexes. Array *columns* stores all the column indexes for the non-zero cells in the row order, i.e., *columns*[*i*] stores the column index of the corresponding cell of *values*[*i*]. Arrays *pointerB* and *pointerE* store the start and end positions of each row in the *values* array, respectively. That is, the positions of the non-zero cells of the *i*th row are between *pointerB*[*i*] and (*pointerE*[*i*]-1).

Given a sparse matrix *M* and a dense vector *V*, we store *M* in the compressed format and compute the multiplication *W*=*M***V*. Our algorithm computes *W* in two steps. First, we scan *values* and

columns once, and compute the partial multiplication results on *values* and *V* into an array, *R*, where *R*[*i*]=(*values*[*i*]**V*[*columns*[*i*]]). This multiplication is done through a gather to fetch the values from the vector *V* at read locations given in the array *columns*. Second, based on *R*, we sum up the partial results into *W*: $W[i]=\sum_{j=pointerB[i]}^{pointerE[i]-1} R[j]$.

When the matrix is large, SpMV is both data- and computation-intensive. We use our multi-pass scheme to reduce the random accesses to the vector. Applying our performance model to SpMV, we estimate the execution time of the gather in Case I.

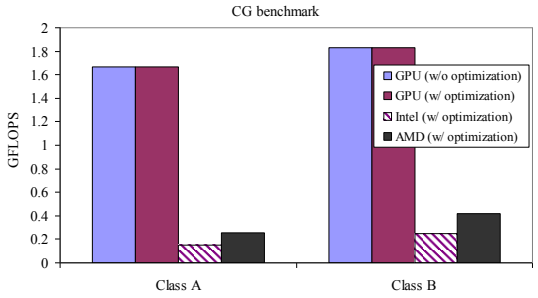

Figure 19. Conjugate gradient benchmark on the GPU and the CPU.

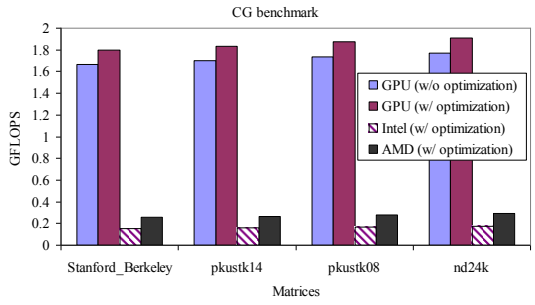

Figure 20. Conjugate gradient benchmark with real-world matrices on the GPU and the CPU.

Figure 19 plots the performance in GFLOPS (Giga Floating Point Operations per Second) of SpMV on the GPU and the CPU. We applied our SpMV algorithm to the conjugate gradient benchmark [5], where SpMV is a core component in the sparse linear solver. The benchmark solves a linear system through calling the conjugate gradient (CG) method *niter* multiple times. Each execution of CG has 25 iterations. Each iteration consists of a SpMV on the matrix and a dense vector, in addition to some vector manipulation. In our experiments, the total execution time of SpMV is over 90% of the total execution time of the CG benchmark on both the GPU and the CPU. There are two classes in the benchmark, classes A and B. The vector sizes are 14 and 75 thousand (i.e., 56KB and 300KB, respectively) for classes A and B, respectively. The *niter* value is 15 and 75 for classes A and B, respectively. The CPU-based SpMV uses the Intel MKL routines, mkl_dcsrmv [3]. The MKL routine does not utilize all the cores on the CPU. We further consider multithreading techniques on the CPU. Suppose the CPU has C_p cores. We horizontally divide the matrix into C_p sub-matrices. Next, each thread performs the multiplication on a sub-matrix and the vector using the mkl_dcsrmv routine. The multithreading technique improves the benchmark around 2X on both AMD and Intel.

Since the vector size is smaller than the cache size on G80, the vector fits into the cache. As a result of this small vector size, the performance of the GPU-based algorithm with optimization is similar to that without. Note that the GPU-based SpMV uses the single precision, whereas the MKL routine uses the double precision. The single-precision computation may have up to 2X performance advantage over the double-precision computation. Nevertheless, the GPU-based algorithm is more than six times faster than the CPU-based algorithm.

We further evaluated our SpMV algorithms using the CG benchmark with real-world matrices. We used the real-world matrices downloaded from the Sparse Matrix Collection [7]. We follow Gahvari's categorization on these matrices [18]: the matrix size is small, medium or large, and the number of non-zero elements per row is low, medium or high. Figure 20 shows the performance comparison for four large real-world matrices, namely pkustk08, pkust14, Stanford_Berkeley and nd24k. These matrices have a similar size, and their non-zero elements per row increase from 11, 98, and 145 to 399. The benchmark configuration is the same as class B. The optimized SpMV with our multi-pass scheme is moderately faster than the one with the single-pass scheme. Moreover, the GPU-based algorithm is over six times faster than the CPU-based algorithm. We also evaluated our algorithms with matrices of different sizes and obtained similar performance results.

5. CONCLUSION AND FUTURE WORK

We presented a probabilistic model to estimate the memory performance of scatter and gather operations. Moreover, we proposed optimization techniques to improve the bandwidth utilization of these two operations by improving the memory locality in data accesses. We have applied our techniques to three scientific applications and have compared their performance against prior GPU-based algorithms and optimized CPU-based algorithms. Our results show a significant performance improvement on the NVIDIA 8800 GPU.

There are several avenues for future work. We are interested in applying our scatter and gather operations to other scientific applications. We are also interested in developing efficient scatter and gather operations on other architectures such as the IBM Cell processor and AMD Fusion. Finally, it is an interesting future direction to extend our algorithms to multiple hosts, e.g., clusters or MPPs without shared memory.

6. ACKNOWLEDGMENTS

The authors thank the shepherd, Leonid Oliker, and the anonymous reviewers for their insightful suggestions, all of which improved this work. Funding for this work was provided in part by DAG05/06.EG11 and 617206 from the Hong Kong Research Grants Council.

7. REFERENCES

[1] CWI Calibrator, http://monetdb.cwi.nl/Calibrator/.

[2] HPF, http://hpff.rice.edu/.

[3] Intel Math Kernel Library 9.0, http://www.intel.com/cd/software/products/asmo-na/eng/266853.htm.

[4] MPI, http://www.mpi-forum.org/docs/.

[5] NAS parallel benchmarks, http://www.nas.nasa.gov/Resources/Software/npb.html.

[6] NVIDIA CUDA (Compute Unified Device Architecture), http://developer.nvidia.com/object/cuda.html.

[7] Sparse matrix collection, http://www.cise.ufl.edu/research/sparse/matrices/.

[8] ZPL, http://www.cs.washington.edu/research/zpl/home/.

[9] M. Abramowitz and I. A. Stegun (Eds.). Stirling numbers of the second kind. In Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables, 9th printing. New York: Dover, 1972.

[10] A. Aggarwal and S. V. Jeffrey. The input/output complexity of sorting and related problems. Communications of the ACM, 31(9):1116–1127, 1988.

[11] D. H. Bailey. Vector computer memory bank contention. IEEE Transactions on Computers, vol. C-36, no. 3 (Mar. 1987), pp. 293-298.

[12] G. E. Blelloch. Prefix sums and their applications. Technical report, CMU-CS-90-190, Nov 1990.

[13] P. Boncz, S. Manegold and M. Kersten. Database architecture optimized for the new bottleneck: memory access. In Proc. of the International Conference on Very Large Data Bases (VLDB), pp 54-65, 1999.

[14] I. Buck. Taking the plunge into GPU computing. In GPU Gems 2, Pharr M., (Ed.). Addison Wesley, Mar. 2005, pp. 509–519.

[15] J. Bolz, I. Farmer, E. Grinspun and P. Schröoder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. ACM Transactions on Graphics 2003, pp. 917-924.

[16] K. Fatahalian, J. Sugerman and P. Hanrahan. Understanding the efficiency of GPU algorithms for matrix-matrix multiplication. In Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2004.

[17] M. Frigo, C. E. Leiserson, H. Prokop and S. Ramachandran. Cache-oblivious algorithms. In Proc. of the 40th Annual Symposium on Foundations of Computer Science, 1999.

[18] H. Gahvari. Benchmarking sparse matrix-vector multiply. Master thesis, Computer Science Division, U.C. Berkeley, December 2006.

[19] N. Galoppo, N. Govindaraju, M. Henson and D. Manocha. LU-GPU: efficient algorithms for solving dense linear systems on graphics hardware. In Proc. of the 2005 ACM/IEEE conference on Supercomputing.

[20] N. Govindaraju, J. Gray, R. Kumar and D. Manocha. GPUTeraSort: high performance graphics coprocessor sorting for large database management. In Proc. of the 2006 ACM SIGMOD international conference on Management of data.

[21] N. Govindaraju, S. Larsen, J. Gray, D. Manocha. A memory model for scientific algorithms on graphics processors. In Proc. of the 2006 ACM/IEEE conference on Supercomputing.

[22] N. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In Proc. of the 2004 ACM SIGMOD international conference on Management of data.

[23] N. Govindaraju, N. Raghuvanshi and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In Proc. of the 2005 ACM SIGMOD international conference on Management of data.

[24] D. Horn. Lib GPU FFT, http://sourceforge.net/projects/gpufft/, 2006.

[25] P. Kipfer, M. Segal, and R. Westermann. Uberflow: A gpu-based particle engine. In Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2004.

[26] A. Lamarca and R. Ladner. The influence of caches on the performance of sorting. In Proc. of the eighth annual ACM-SIAM symposium on Discrete algorithms, 1997.

[27] E. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In Proc. of the 2001 ACM/IEEE conference on Supercomputing.

[28] K. Moreland and E. Angel. The FFT on a GPU. In Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2003.

[29] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn and T. J. Purcell. A survey of general-purpose computation on graphics hardware. Computer Graphics Forum, Volume 26, 2007.

[30] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2003.

[31] S. Sengupta, M. Harris, Y. Zhang, J. D. Owens. Scan primitives for GPU computing. In Proc. of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, 2007.

[32] J. Vitter. External memory algorithms and data structures: Dealing with massive data. ACM Computing Surveys, 209-271, 2001.

[33] S. Williams, J. Shalf, L. Oliker, P. Husbands and K. Yelick, Dense and sparse matrix operations on the Cell processor. May, 2005. Lawrence Berkeley National Laboratory. Paper LBNL-58253.

[34] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In Proc. of the 1991 ACM/IEEE conference on Supercomputing.