HDGEMM: a Hybrid DGEMM library on a Heterogeneous Architecture with CPU and GPU

Abstract

In a heterogeneous architecture, data transfer between CPU and GPU is a performance critical factor in real applications. In this paper we present a Hybrid Double-precision GEneral Matrix Multiplication (HDGEMM) library, which overlaps data transfer with DGEMM computation on a heterogeneous system of CPU and ATI GPU. With respect to the effect of software pipelining on reducing the overhead of data transfer, we develop three pipelining DGEMM algorithms: double buffering, data reuse and data placement. On ATI Radeon™ HD5970, HDGEMM achieves 758GFLOP/s with an efficiency of 82%, which is more than 2 times higher than the latest AMD library ACML-GPU v1.1.2. It also achieves 844GFLOP/s with an efficiency of 80% on a heterogeneous system of Intel Westmere EP and ATI Radeon™ HD5970. Further, we measure the intra-node scalability of HDGEMM library. When scaling to multiple CPUs and GPUs, it is found that the efficiency decreases with more computing units, though HDGEMM performance increases progressively. We perform a comprehensive analysis on the influencing factors, and identify that it is the resource contention (especially PCIe and system memory contention) that harms HDGEMM scalability on a heterogeneous system.

Categories and Subject Descriptors F.2.1 [Numerical Algorithms and Problems]: Computations on matrices; C.1.2 [Multiple Data Stream Architectures (Multiprocessors)]: Single-instruction-stream multiple-data-stream processors

General Terms Algorithm, Performance

Keywords high performance computing; GPU; CAL; matrix-matrix multiplication; library

1. Introduction

Double-precision GEneral Matrix Multiplication (DGEMM) is a performance critical kernel in scientific and engineering applications, such as BLAS [1] and HPL [3] which ranks supercomputers in the world. Since its performance depends highly on underlying hardware, all processor vendors provide highly optimized DGEMM implementations (e.g. Intel MKL and AMD ACML) on their own processors. However, GPU offers more than an order of magnitude speedup of peak floating-point computing power over conventional processors. For instance, ATI Radeon™ HD5970, which employs two GPU chips codenamed "Cypress", reaches 928GFLOP/s and NVIDIA Tesla C2070 offers 515GFLOP/s in double-precision. With the popularity of CPU-GPU heterogeneous architectures and the higher computing power of GPU, it is necessary to optimize DGEMM using GPUs. Moreover, DGEMM is compute-intensive and exhibits regular memory access patterns, which are supposed to be well suited to GPU. As a matter of fact, there is a lot of work accelerating DGEMM on GPUs [10-20, 23-24]. Note that most of the previous work only focuses on GPU kernel optimizations, assuming that data are already resident in GPU on-board memory. However, in real applications all data are initially located in CPU system memory before GPU begins calculating. There exists data transfer between CPU and GPU before and during GPU calculation, when the problem size is too large to completely fit into GPU memory.

In a heterogeneous CPU-GPU system, data movement through memory hierarchy plays an important role on DGEMM performance. From the system view, there are two levels in the memory hierarchy including CPU system memory and GPU local memory. Before GPU kernel fetches data from its local memory for computation, data transfer between CPU and GPU through PCIe bus is needed. Note that the bandwidth of GPU local memory is 256GB/s on HD5970 while PCIe2.0 only provides a peak bandwidth of 8GB/s in each direction. Although DGEMM is compute-intensive, the bandwidth gap still becomes a bottleneck for its performance on a heterogeneous system. Nakasato [14] showed that the efficiency of DGEMM decreases from 85% to 55% when the overhead of data transfer between CPU and ATI Cypress GPU is included. DGEMM efficiency drop is also observed in ACML-GPU v1.1.2 [5], a library released by AMD to accelerate GEMM functions on a heterogeneous CPU-ATI GPU system.

This phenomenon raises our question: *how does memory hierarchy really affect DGEMM performance on a current heterogeneous CPU-GPU architecture?* Although there is some previous work optimizing DGEMM on GPUs, few available materials reported quantitative analysis about this issue. A quantitative analysis of each memory level will be helpful to develop program optimization approaches for building a high performance DGEMM library and to give advice on hybrid architecture in the future. Currently, one approach to build a hybrid CPU-GPU system is to integrate multiple CPUs and GPUs on the same board inside a single computing node. As we know, the potential for inter-node scalability of DGEMM is high, because we can divide the matrix into smaller ones and allocate them to each node. Thus the sub-matrices can be computed independently without sharing resources. However, when scaling inside a node, resource contention may harm DGEMM performance. So, *what about HDGEMM behavior when scaling to multiple CPUs and GPUs inside a computing node?* We will give a detailed analysis on our HDGEMM library scalability, and disclose the influencing factors.

In this paper we address the above issues by investigating a heterogeneous system of multi-core CPU and ATI Cypress GPU. We resort to alternative algorithmic solutions to reduce the overhead of data movement through memory hierarchy on current heterogeneous CPU-GPU architecture. Specifically, we make three main contributions in this paper:

- We quantitatively analyze DGEMM performance on a heterogeneous architecture counting data movement between CPU and GPU. The analysis identifies the undercover data transfer, and shows that data transfer comprises more than 40% of the total time. This high

overhead has considerable impact on DGEMM performance. (Section 2)

- We build a new hybrid DGEMM library (HDGEMM) on a heterogeneous architecture, which utilizes a new pipelining algorithm intended for large scale matrices. This algorithm is based on three incremental optimization approaches: double buffering, data reuse and data placement. Experimental results show that HDGEMM achieves 408 GFLOP/s performance with an efficiency of 88% on one Cypress GPU. Compared with AMD's ACML-GPU v1.1.2, HDGEMM improves the performance by more than 2 times. It also achieves 758GFLOP/s with an efficiency of 82% on ATI Radeon™ HD5970, and 844GFLOP/s with 80% efficiency on the heterogeneous system of Intel Westmere EP and ATI Radeon™ HD5970. (Section 3)
- Finally, we analyze the intra-node scalability of HDGEMM when scaling to multiple CPUs and GPUs. Though HDGEMM performance increases progressively, the efficiency decreases with more computing units. According to our analysis, we find that the major bottleneck is resource contention (especially PCIe contention and system memory contention), and it is difficult to further avoid the overhead by a pure software solution. Moreover, three observations are presented to give advice on relaxing the reosurce contention in the future. (Secion 4)

Though we use a heterogeneous system with Intel CPU and ATI GPU in this paper, our pipelining algorithm is also suited to other CPU and PCIe attached accelerator platforms with correspond sub-matrices size. When applying the pipelining algorithm to NVIDIA GPU, because of its pinned memory the data buffering optimization is not needed.

The rest of the paper is organized as follows. In Section 2, we quantitatively analyze the un-optimized DGEMM on the heterogeneous architecture with one Cypress GPU. Three incremental pipelining algorithms are proposed in Section 3. Section 4 gives the performance results of our HDGEMM library and detailed analysis about its intra-node scalability. Related work is presented in Section 5, and conclusions in Section 6.

2. Background

Although our paper focuses on ATI Cypress GPUs, the optimizations proposed will also be directly applicable to other ATI GPUs. ATI Radeon™ HD5870 card contains one Cypress chip, while ATI Radeon™ HD5970 card integrates two Cypress chips. This section highlights several important features of Cypress, especially for memory hierarchy in ATI CAL software layer [4]. Since the DGEMM routine in ACML-GPU library is used as a baseline program for performance optimization, we profile its execution without pipelining to motivate our optimizations.

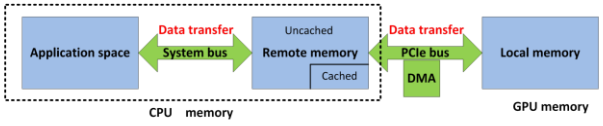

Figure 1. Memory hierarchy in CAL system between CPU and ATI GPU

2.1 Cypress GPU

One Cypress GPU chip integrates a controller unit named ultra-threaded dispatch processor, multiple computing units, memory controllers and DMA engines. The Cypress chip microarchitecture is customized with single-instruction-multiple-data (SIMD) and very long instruction word (VLIW) for extremely high throughput of floating-point operations. We do not describe the detailed microarchitecture since this paper does not focus on kernel optimizations (the kernel used in our work has approached an efficiency of more than 80%). We suggest [14] to readers for more details. Totally, ATI Radeon™ HD5970 offers a peak performance of 2.32TFLOP/s in single-precision operations at 725Mhz core frequency. Double-precision floating point operations are performed one-fifth of the single precision, which produces 464GFLOP/s in double-precision at 725Mhz.

Figure 1 depicts the memory hierarchy in CAL system on a heterogeneous CPU-ATI GPU architecture, i.e. application space, remote memory and local memory. At the first glance, it seems a common memory hierarchy with gaps of latency and bandwidth. Below we will detail the features related to performance optimization in CAL system for HD5970 GPU:

- CAL system abstracts physical memory into local and remote memory. Local memory corresponds to the high-speed video memory located on graphics board. Remote memory corresponds to the one that is not local to GPU device but still visible to it (i.e. some regions in host memory). Both remote and local memory can be directly accessed by GPU kernel. That is, a GPU kernel can dump data from its registers into remote memory using store instructions. But such a store operation results in poor performance, because access to remote memory is slower and has higher latency compared with local memory. Moreover, remote memory is partitioned into cached and uncached parts. For example, in ATI Stream SDK2.2 the CAL system reserves 500MB cached memory and 1788MB uncached memory on HD5970 (the main experimental platform in this paper). Therefore, *the application performance is sensitive to which memory region for the shared data is selected between CPU and GPU*.
- A typical CAL application initializes data in CPU application space, which is in host memory. There is a two-step data transfer process: between application space and remote memory, between remote memory and GPU device (local) memory. One optimization technique is the use of system pinned memory, where the application stores data into the memory that can be directly used for DMA transfer, so that the copy inside host memory will be skipped. It has been proven efficient to improve the performance of some applications while using CUDA [7]. In CAL, the equivalent technique is the use of remote memory, but there are some constraints such as the limited size. Thus, *it is necessary to orchestrate the two-level data transfer with proper memory management (i.e. pipelining) to shorten the time needed*.

2.2 DGEMM

In this section we describe an algorithmic framework for large scale DGEMM on a heterogeneous CPU-ATI GPU system, with initial data resident in the CPU application space. DGEMM calculates C := alpha *A * B + beta * C, where A, B and C are m*k, k*n, m*n matrices, respectively. Since in most of DGEMM applications the three matrices are too large to be held in GPU memory, they are partitioned into multiple sub-matrices to perform multiplication block-by-block. We assume the three matrices are partitioned as A={A1, A2, … , Ap}, B={B1, B2, … , Bq}, C={C1, C2, … , Cpxq}, where p and q depend on GPU memory size. For simplicity of presentation, we take p=2 and q=2 for example and the matrix multiplication is illustrated in Figure 2.

The partition results in four independent sub-matrices of C which are calculated in parallel: C1=A1*B1, C2=A1*B2,

C3=A2*B1, C4=A2*B2. For each sub-matrix multiplication we load its dependent sub-matrices A and B into GPU memory, then DGEMM kernel may further divide them into smaller ones for faster multiplication [8-12] (i.e. cache blocking and register blocking). Based on the memory hierarchy in ATI CAL system, DGEMM program is described in Algorithm 1. Remote memory acts as a shared space between CPU and GPU, and data in it will be shared by application space and GPU local memory (Line 1). In the pseudo-code of Algorithm 1, there are two processes for loading data to GPU local memory (*load1*, *load2*) and one process for storing data back to CPU application space (*store*). In fact, line 5 which represents DGEMM kernel execution implicitly includes another store operation -- storing the results of C sub-matrices from registers to remote memory.

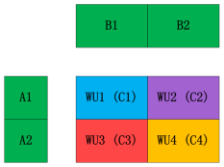

Figure 2. An example of Workunits split when p=2 and q=2

Partition: $A=\{A_1, A_2, \ldots, A_p\}$, $B=\{B_1, B_2, \ldots, B_q\}$, $C=\{C_1, C_2, \ldots, C_{pxq}\}$
Work unit: $WU=\{C_1=A_1*B_1, C_2=A_1*B_2, \ldots\}$
//
1. bind remote memory for sub-matrices A,B,C
2. **for** each workunit wu_i **do** *//i=1,2,…,pxq*
//load1
3. copy both A_i and B_i from application space to remote memory
//load2
4. copy both A_i and B_i from remote memory to local memory
//mult
5. calculate C_i on GPU device and directly output it to remote memory
//store
6. copy C_i form remote memory to application space (also multiplied by *beta*)
7. **endfor**

Algorithm 1. Initial DGEMM implementation

To get a baseline for performance comparison, we implemented Algorithm 1. This section presents a detailed profiling of its execution on a heterogeneous CPU-GPU architecture. Figure 3 plots the time percentage of each step in Algorithm 1. As the time distributions of different problem scales show similar behavior, we take k=2048 as an example, and give the matrix order m (=n) in x-axis. We learn that *mult* kernel occupies the most percentage (more than 70%) while the sum of the three data transfer steps is less than 30%. Intuitively, the overhead of data transfer can be hidden by overlapping with the kernel execution. In order to find out an approach to achieve this, we classify the resources involved in the algorithm into CPU + Host Memory, GPU and PCIe Bus. The operations on these three classifications can proceed in parallel. Table 1 outlines the resource usage in each step of Algorithm 1. Both *load1* and *store* consume the resources of CPU + Host Memory, and *load2* only needs PCIe bus to transfer data. The *mult* kernel calculates DGEMM on GPU, and then outputs its results to remote memory through PCIe Bus. Based on the observation of resource usage in each step, it is straightforward to implement a software pipelining algorithm to overlap the data transfer (*load1*, *load2*, *store*) with *mult* kernel. In previous work, ACML-GPU overlaps some data transfer steps with a *mult* step. Yang et.al [19] implemented a pipelining algorithm that overlaps *load1* with *mult*. However, as shown in their literature and our experiments in Section 4, simple pipelining algorithm improves DGEMM performance merely (about 20%). In fact, the most time-consuming part of Algorithm 1 is *mult* kernel. As shown in Algorithm 1, it directly dumps results from registers to remote memory, which implies another data transfer through PCIe bus. Therefore, the data transfer becomes a performance bottleneck because: i) PCIe bandwidth is much lower than memory bandwidth of either CPU or GPU; ii) The size of transferred data in *mult* may be larger than that in *load2*. Take DGEMM in LINPACK benchmark for example, A, B and C are m*k, k*n, m*n matrices respectively, where k is much less than both m and n. Thus the size of resulting C matrix (m*n) is larger than the size of A and B (k*(m+n)). Besides, as described in Section 3 we can reduce the frequency of data transfer in *load1* and *load2* by exploiting data reuse. Therefore, we refine the pipelining algorithm between workunits to achieve better overlap between floating-point operations and data transfer operations.

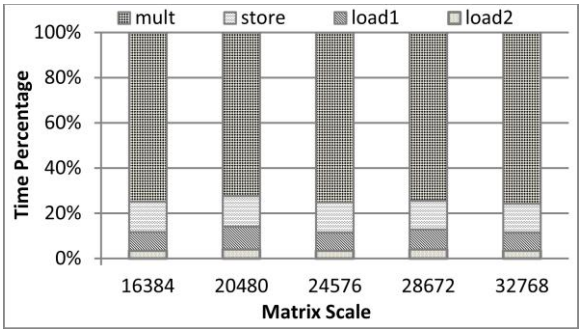

Figure 3. Time composition in initial DGEMM implementation

Table 1. Resource allocation in each step of Algorithm 1

	CPU + Host Memory	GPU	PCIe Bus
Load1	■		
Load2			■
Mult		■	■
Store	■		

3. Pipelining Algorithm

Software pipelining is a common technique to overlap computation with memory operations. As shown in Algorithm 1, there are three explicit memory operations (*load1*, *load2*, *store*) for transferring data between CPU and GPU, and one multiplication operation (*mult*) which directly outputs results to remote memory. Therefore, the pipelining algorithm needs to perform *mult* in parallel with the three explicit memory operations. Since the sum of *load1*, *load2*, *store* execution time is much less than that of *mult* (Figure 3), their overhead can be hidden by the pipelining execution at the level of workunits. However, there exist resource conflicts to CPU memory between *load1* and *store*, and conflicts to PCIe bus between *load2* and *mult*. The resource conflicts lead to delay in the pipeline and decrease DGEMM performance. We exploit three optimizations to avoid the resource conflicts in the next three sub-sections: double buffering, data reuse and data placement.

3.1 Double Buffering

The *store* for writing back C matrix performs both data transfer and a small portion of floating-point calculations (beta * C) which

also consumes CPU resources. One way to hide its overhead is to implement a pipeline at the level of workunits. For example, the *mult* of workunit i+1 can be launched when the *store* in workunit i is issued. However, there are two problems in such a pipeline. First, *load1* has a resource conflict with *store* because both of them are moving data between application space and CAL remote memory. The resource conflict decreases the efficiency of the pipeline. Secondly, the size of remote memory is limited, especially for the cached remote memory. The *store* is also executed on CPU (beta*C), while data for *load1* is not reused due to large matrix scale. It's better to hold the output C matrices using cached remote memory. The small size of cached remote memory limits the number of concurrent workunits during the execution of pipeline.

A better strategy to overcome the above shortcomings is to exploit a fine-grained pipelining execution inside a workunit, rather than between multiple workunits. Algorithm 2 shows the fine-grained pipelining execution. The algorithm further partitions sub-matrices of C into many blocks, which are computed in a pipelined way. In the meantime, more than one buffer is needed to implement the pipeline. Considering the limited size of cached remote memory, we allocate two buffers in it and implement a double buffering algorithm. For each loop j in line 6-9 of Algorithm 2, *store* dumps one buffer of a block $C_{i,j}$ into application space while *mult* is calculating the next $C_{i,j+1}$ and filling it into the other buffer. During the execution of this pipeline, the two buffers are used by *mult* and *store* in an interleaved manner. Since *mult* kernel is executed on GPU device in a non-blocking way, it proceeds in parallel with *store* in every iteration.

Partition: $A=\{A_1, A_2, \ldots, A_p\}$, $B=\{B_1, B_2, \ldots, B_q\}$, $C=\{C_1, C_2, \ldots, C_{pxq}\}$
Work unit: $WU=\{C_1=A_1*B_1, C_2=A_1*B_2, \ldots\}$
C_{ij}*: the sub-matrices of* C_j
//
1. bind remote memory for sub-matrices A,B,C
//the for-loop is pipelined
2. **for** each workunit wu_i **do** *//i=1,2,…,pxq*
//load1
3. copy both A_i and B_i from application space to remote memory
//load2
4. copy both A_i and B_i from remote memory to local memory
//mult
5. calculate $C_{i,1}$ on GPU device and output it to remote memory
6. ***for*** each block $C_{i,j}$ **do** //j=2,3…
//store
7. copy $C_{i,j-1}$ form remote memory to application space (also multiplied by *beta*)
//mult
8. calculate $C_{i,j}$ on GPU device and output it to remote memory
9. ***endfor***
//store
10. copy the last $C_{i,j}$ form remote memory to application space (also multiplied by *beta*)
11. **endfor**

Algorithm 2. DGEMM with Double Buffering

3.2 Data Reuse

Through data buffering algorithm, we successfully make *mult* and *store* execute in parallel inside one workunit. This algorithm decreases the resource conflicts between *load1* and *store*. As we know sub-matrices of A and B cannot be reused because of the large matrices scale. Fortunately, we can still exploit data reuse between two consecutive workunits.

Take the example in Figure 2, if we schedule the execution of workunits in an order of WU1={C1=A1*B1}, WU2={C2=A1*B2}, WU3={C4=A2*B2}, WU4={C3=A2*B1}, every two consecutive workunits reuse one of the two input matrices. In order to exploit such data reuse to further reduce the overhead of resource conflicts, two extra steps should be performed. First, a pre-processing is used to construct a queue, which defines the execution order of the workunits for exploiting data reuse. We first divide matrix C to a pack of matrix bars, named with an integer (i=0,1…). A matrix bar is further divided into different blocks from the bottom up when i%2 = 0, while from the top down when i%2 = 1. In this way, we push the divided matrix blocks into the queue in a wriggled way, traversing from top to bottom in one bar and going the other way in the next bar. Secondly, two indicators are set as addresses of matrices A and B in the current workunit (line 3 and 4), so that the next workunit avoids loading the same matrix blocks again. Algorithm 3 outlines the pipelining algorithm with data reuse.

Partition: $A=\{A_1, A_2, \ldots, A_p\}$, $B=\{B_1, B_2, \ldots, B_q\}$, $C=\{C_1, C_2, \ldots, C_{pxq}\}$
Work unit: $WU=\{C_1=A_1*B_1, C_2=A_1*B_2, \ldots\}$
C_{ij}*: the sub-matrices of* C_j
//
1. bind remote memory for sub-matrices A,B,C
//pre-processing
Allocate workunits in a wriggled way
//the for-loop is pipelined
2. ***for*** each workunit wu_i **do** *//i=1,2,…,pxq*
//load1
3. copy either A_i or B_i from application space to remote memory according to the indicators
//load2
4. copy either A_i or B_i from remote memory to local memory according to the indicators
//mult
5. calculate $C_{i,1}$ on GPU device and output it to remote memory
6. ***for*** each block $C_{i,j}$ **do** //j=2,3…
//store
7. copy $C_{i,j-1}$ form remote memory to application space (also multiplied by *beta*)
//mult
8. calculate $C_{i,j}$ on GPU device and output it to remote memory
9. ***endfor***
//store
10. copy the last $C_{i,j}$ form remote memory to application space (also multiplied by *beta*)
11. ***endfor***

Algorithm 3. DGEMM with Data Reuse

3.3 Data Placement

For better data reuse, both A and B are temporarily stored in GPU local memory. As for the remote memory regions of both A and B, they are in uncached type because (i) no floating-point calculation is executed by CPU; (ii) cached remote memory is too small to accelerate *load1* effectively. Since sub-matrices of C in all workunits are independent with each other, there is not data reuse between them. It seems reasonable that they are directly stored in remote memory rather than GPU local memory, as the size of local memory is very limited on GPU device. Therefore, in the above three algorithms sub-matrices of C are stored in cached remote memory for better CPU performance; that gives higher speed memory copy for double buffering and faster floating points operations (multiplied by beta).

Owing to the pipelining execution, memory transfer operations (*load1*, *load2*, *store*) are overlapped with the kernel multiplication (*mult*). So far, the execution time of the pipeline is determined by *mult* kernel. Therefore, one optimization approach left is to reduce the time consumed by *mult*. Note that *mult* contains memory operations which directly dump C values from GPU registers to remote memory. Our optimization strategy is to add an extra

phase to the pipeline, so that the output operations to remote memory will also be overlapped by GPU kernel computation.

In order to construct a new phase in the pipeline, the operations for outputting sub-matrices of C to remote memory are separated from *mult* kernel, while the kernel output is updated in GPU local memory. Thus, as shown in Algorithm 4, the original *mult* is further split into two phases: *mult1* and *store1*. The *store1* transfers C results from local memory to remote memory. With respect to resource occupancy, *mutl1* is executed completely on GPU device while *store1* is performed by DMA engine. Due to the asynchronous execution of kernel and DMA operations, they can be executed in parallel within local memory similar to the double buffering strategy. For more clarity, we will use *store2* instead of the *store* operation in the following sections.

Partition: $A=\{A_1, A_2, \ldots, A_p\}$, $B=\{B_1, B_2, \ldots, B_q\}$, $C=\{C_1, C_2, \ldots, C_{pxq}\}$
Work unit: $WU=\{C_1=A_1*B_1, C_2=A_1*B_2, \ldots\}$
C_{ij}: *the sub-matrices of* C_j
//
1. bind remote memory for sub-matrices A,B,C
//pre-processing
Allocate workunits in a wriggled way
//the for-loop is pipelined
2. ***for*** each workunit wu_i **do** *//i=1,2,…,pxq*
//load1
3. copy either A_i or B_i from application space to remote memory according to the indicators
//load2
4. copy either A_i or B_i from remote memory to local memory according to the indicators
//mult
5. ***DMAPipeline***$(C_{i,1})$
6. ***for*** each block $C_{i,j}$ **do** //j=2,3…
//store2
7. copy $C_{i,j-1}$ form remote memory to application space (also multiplied by *beta*)
//mult
8. ***DMAPipeline***$(C_{i,j})$
9. ***endfor***
//store2
10. copy the last $C_{i,j}$ form remote memory to application space (also multiplied by *beta*)
11. ***endfor***

Algorithm: DMAPipeline$(C_{i,j})$
$C_{i,j,k}$: *the sub-blocks of* $C_{i,j}$
//
//the for-loop is pipelined
//mult1
1. calculate $C_{i,j,1}$ in local memory
2. ***for*** each sub-block $C_{i,j,k}$ **do** //k=2,3…
//store1
3. DMA transfer $C_{i,j,k-1}$ from local memory to remote memory
//mult1
4. calculate $C_{i,j,k}$ in local memory
5. ***endfor***
//store1
6. DMA transfer the last $C_{i,j,k}$ from local memory to remote memory

Algorithm 4. DGEMM with Data Placement

So far, we have a five-stage (*load1*, *load2*, *mult1*, *store1*, and *store2*) pipeline in our hybrid DGEMM library. In Table 2, we depict the new resource allocation in HDGEMM. The *mult1* kernel no longer needs PCIe bus and system memory, and only uses the GPU device. Therefore, without PCIe conflicts, *mult1* kernel could execute in parallel with *load2* and *store1*. Algorithm 4 not only gives a finer pipeline and alleviates the delay generated by resource conflicts, but also allows for a faster kernel implementation. We further show a schematic sketch of the pipeline in Figure 4, where resource conflicts in mini-steps between different workunits are still included. We distinguish the five steps with different colors. The bars inside a *mult1* block represent the data transfer steps (*load1*, *load2*, *store1* and *store2*), which are overlapped by *mult1* kernel. As shown in this figure, except for prologue and epilogue of the pipeline, most of data transfer in Algorithm 4 can be fully overlapped.

Table 2. Resource allocation in optimized DGEMM (Algorithm 4)

	CPU+Memory Bus	GPU	PCIe Bus
Load1	■		
Load2			■
Mult1		■	
Store1			■
Store2	■		

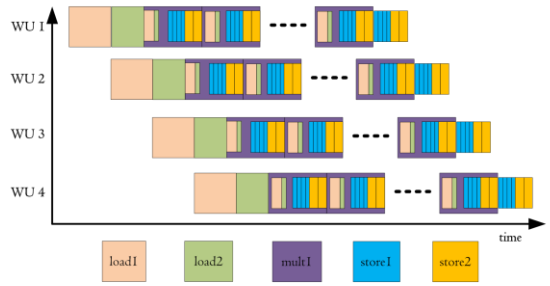

Figure 4. Our optimized DGEMM pipeline sketch

Table 3. System configuration of the experimental platform

processors		Intel Xeon X5650	ATI Radeon™ HD5970
model		Westmere EP	Cypress
frequency		2.66Ghz	725Mhz
#chips		2	2
DP		128GFLOP/s	928GFLOP/s
DRAM	*type*	DDR3 1.3Ghz	GDDR5 1.0Ghz
	size	24GB	2GB
	bandwidth	31.2GB/s	256GB/s
PCIe2.0		x16, 8GB/s	
programming		icc + openmpi	ATI Stream SDK 2.2

4. Experiment Results and Analysis

4.1 Experimental Setup

The experiments are performed on the heterogeneous system composed of two Intel Xeon 5650 CPUs and one ATI Radeon™ HD5970 GPU card. Table 3 summarizes the configuration parameters of the experimental platform. The multi-core CPU provides peak double-precision performance of 128GFLOP/s. The CPU memory system is configured with a size of 24GB and an aggregated bandwidth of 31GB/s. The GPU containing two Cypress chips provides a peak double-precision computational capability of 928 GFLOP/s. The GPU memory size is 2GBytes and its memory bandwidth reaches 256GB/s. The total peak performance of the heterogeneous system is 1056GFLOP/s.

In order to perform a comprehensive experimental analysis, we implement several programs which incrementally adopt the optimization strategies proposed in Section 3. To simplify the presentation, we define some notations referring to different versions.

- DB: The program implements Algorithm 2. It uses double buffering strategy to hide the overhead caused by writing resultant C matrices from remote memory to application space. The program allocates two buffers in cached remote memory, the size of which is determined by matrices size in GPU local memory.
- *DR*: This program implements the pipelining Algorithm 3, which is based on the version of *DB* and improved by overlapping the load operations of input matrices. An important optimization to the pipelining algorithm is to exploit data reuse while reading the input matrices.
- *DP*: In addition to double buffering and data reuse, this program implements Algorithm 4 which focuses orchestrating the data placement through the memory hierarchy of CAL system. A critical optimization is to select a better placement for the output C matrices and take the advantages of DMA to design a more efficient pipeline.
- *HB*: The above three programs only exploit the computational capability of GPU. In this program we further implement a hybrid DGEMM, where CPU and GPU perform multiplications cooperatively. This is our final DGEMM library. The matrices are partitioned between CPU and GPU, and we adopt the strategy proposed in [19] for workload balance between them. In our experiments, we use two processes, and each of them uses a pair of one CPU and one GPU chip.

Note that the four programs represent four incremental optimization strategies, and each program adds another optimization strategy in the following order of DB<DR<DP<HB. The source of ACML-GPU [5] library that implements DGEMM on GPU is available with the package. We use this code as our initial experiment and a baseline to evaluate the described optimizations in this paper.

Table 4. DGEMM matrices scale of *m,n,k* and the size of A,B,C matrices in MBytes used in the following experiments.

k	*m=n*				
	16384	20480	24576	28672	32768
1536	304	460	648	868	1120
2048	320	480	672	896	1152
4096	384	560	768	1008	1280

Table 4 shows matrix order (m,n,k) in our experiments, and the size of the all the three matrices in MBytes. The size of m and n are kept equal because the difference between them has little effect on DGEMM performance. The size of k determines the amount of data reuse during reading the input matrices, which has a significant impact on performance. Therefore, three values of k are used to represent three data sets. In the sections below, we give the performance values for a particular k size by calculating the average performance over all the five values of m(=n). As for detailed profiling, we always take k=2048 for example in default, and the matrix scale in x-axis represents different m (=n) values.

4.2 Results

Firstly, we report the overall performance of our HDGEMM library on the Intel Westmere EP and ATI Radeon™ HD5970 heterogeneous system. The baseline program for performance comparison is ACML-GPU library v1.1.2. Figure 5 plots the HDGEMM performance and efficiency with different data sizes, as given in Table 4. Our HDGEMM with two GPU chips (HB-2GPU) achieves a maximal performance of 844GFLOP/s and an efficiency of 80% for (m,n,k) = (16384, 16384, 4096). Since ACML-GPU library doesn't have a hybrid CPU-GPU cooperation like HDGEMM, in fairness we add the optimized DGEMM implementation (DP) in Figure 5 for comparison. DP implemented on two GPU chips (DP-2GPU) reaches a maximal performance of 758GFLOP/s and an efficiency of 82% for (m,n,k) = (16384, 16384, 4096). The results show that DP-2GPU improves DGEMM performance by more than 2.0 times on average over ACML-GPU library. HB-2GPU further improves performance by 10%-20%. When k is fixed, all the three programs show an increase in both performance and efficiency with larger matrices. There are few cases with abnormal performance (e.g. m=n=10240). It is because the problem scale is not multiples of the optimal blocking size of data transfer and kernel execution. When comparing the average performance among different ks, the performance improvement of DP over ACML-GPU drops as the matrix scale becomes larger. When k is 1536, 2048 and 4096, the speedup is 2.9X, 2.1X, and 1.9X respectively. This phenomenon is expected, because the ratio of data transfer to kernel execution decreases with larger problem scales. However, it is the data transfer that our pipelining optimizations show the efficacy better. We would like to point out that it is relatively easier to achieve higher efficiency on GPUs for larger datasets, compared with smaller datasets. As for larger datasets, the calculation part will make up the major proportion of the DGEMM implementation, and the effect of data transfer will be weakened. As shown in Figure 5, we also learn HB-2GPU has more performance improvement with larger matrices size. That's because CPU gains better performance, and enhances the whole performance of HDGEMM, when matrices size becomes larger.

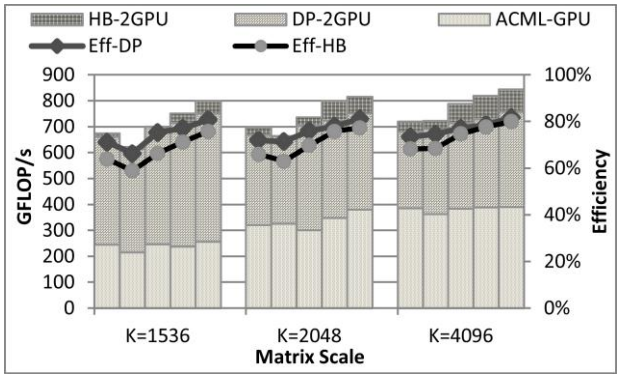

Figure 5. Our optimized DGEMM Performance and efficiency with two processes (*m,n* from Table 4)

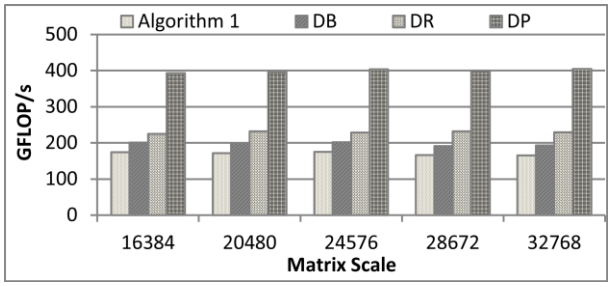

Figure 6. Performance improvement of each optimization approach

Secondly, we evaluate the effect of the three optimization strategies described in Section 3. In order to isolate other noises (e.g. contention on bandwidth, which will be reported in the next section) in the system, we perform the experiments on only one

GPU chip and do not assign any floating-point computing workload of matrix multiplication to CPU, which only performs data transfer cooperatively. Figure 6 plots the performance increments achieved by double buffering (DB), data reuse (DR) and data placement (DP), respectively. Compared with Algorithm 1, double buffering improves 16% performance by pipelining *store2* inside a workunit. Data reuse improves performance by another 18%, overlapping load operations (*load1*, *load2*) with the multiplication kernel between different workunits. Finally, the performance is significantly improved by 74% using data placement, which refines the initial *mult* kernel and leverages DMA engine to pipeline the operations of writing back the resulting C matrices. DP implementation achieves 408 GFLOP/s with an efficiency of 88% on one Cypress GPU.

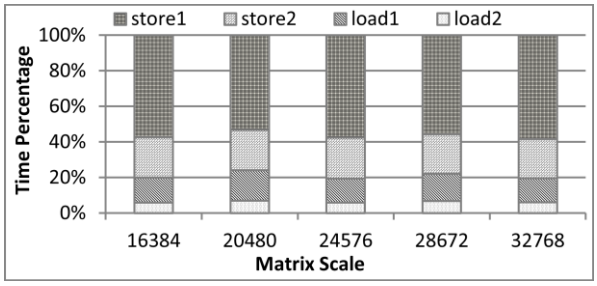

Figure 7. Percentage of the four data transfer steps in optimized DGEMM

In Section 2, Figure 3 shows that the three data transfer steps (*load1*, *load2*, and *store*) occupy less than 30% of the total execution time, as the statistics does not include the implicit data transfer (*store1*) time in *mult* kernel. Our optimization refines the pipelining algorithm by separating *store1* from *mult* kernel. Obviously, our approach is closer to the nature of the pipelining design. In fact, after counting the data transfer in *mult* kernel, the total data transfer occupies more than 40% of the total execution time. We further give the time percentage of each data transfer step (*load1*, *load2*, *store1*, and *store2*) in Figure 7. It only includes the four data transfer steps without *mult1* kernel, and y-axis represents the ratio of the time of each data transfer step to the total data transfer time. From this figure, we see that the performance improvement by each optimization in Figure 6 is in accordance with the data transfer time distribution. It demonstrates that our optimizations make full use of pipelining. Besides, there is an additional performance improvement to data placement because our implementation optimizes multiplication kernel as well. We also learn from Figure 6 that our optimized DGEMM performance is quite steady through all the matrix sizes. This property lays a good foundation for the good scalability of our HDGEMM library on multiple CPUs and GPUs. However, this trend on one GPU chip is different from that shown in Figure 5, which reports the overall performance on a heterogeneous system. We will discuss this phenomenon in the next section.

4.3 Analysis

Usually, DGEMM in CPU's math library can achieve a maximal efficiency of more than 90%. Our HDGEMM library on the heterogeneous system also achieves the maximal efficiency of 82%, counting data transfer between CPU and GPU. In this section we will investigate (i) How much optimization room is left beyond our pipelining optimization on such a heterogeneous architecture, and (ii) What can be said about the intra-node scalability of our HDGEMM library when scaling to multiple CPUs and GPUs.

4.3.1 Performance Gap

In the five phases of the execution pipeline, *mult1* kernel determines the highest performance the DGEMM may achieve. N.Nakasato [14] optimized DGEMM kernel performance and reported the highest efficiency of 87% on HD5870 including one Cypress chip. We adopt Nakasato's strategies to optimize the original kernel in ACML-GPU library. However, we improve the implementation by using image read/write addressing mode instead of global buffer addressing mode, and achieve a higher efficiency of 94% on one Cypress chip. We have also seen better kernel performance compared to the AMD Accelerated Parallel Processing Math Libraries (APPML) v1.4 [6]. Note that APPML provides GPU-only DGEMM kernels, while the focus of our work is to provide an optimized Hybrid DGEMM library on a heterogeneous CPU-GPU system.

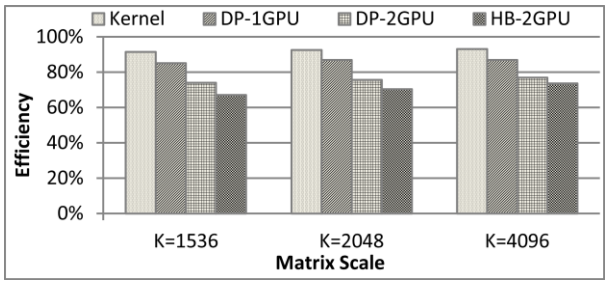

Figure 8. The efficiency of our optimized DGEMM when scaling to multiple CPUs and GPUs

Figure 8 plots the efficiency comparison among *mult1* kernel, DP on one GPU chip, DP on two GPU chips, and hybrid DGEMM on two CPUs and GPU chips. For each set of experiments, we show its average efficiency. One time execution of DGEMM calls *mult1* kernel multiple times. We measure the kernel's performance in GFLOP/s each time and calculate the average value as its final GFLOP/s in the figure. Our optimized kernel only reads and writes GPU local memory, thus its performance has no relation to CPU resources. As shown in this figure the kernel's efficiency is over 90% (the best one is 94%), which is comparable to the performance of the optimized DGEMM library on CPU. The difference between the kernel and DP is whether the data transfer between CPU and GPU is included. The optimized programs proposed in this paper use software pipelining to overlap the overhead of data transfer. Experimental results show that DP-1GPU decreases 6% performance from kernel due to the data transfer. There are two reasons for the performance drop. First, the prologue and epilogue of the pipeline cannot be hidden, which occupy around 3% of the total DGEMM execution time. Secondly, as shown in Table 2, there are still intrinsic resource conflicts inside the pipeline. During the pipeline execution, there may be system memory contention between *load1* and *store2*, and PCIe bus contention between *load2* and *store1*. Apart from the two factors, DP-1GPU almost achieves the best performance on one GPU chip.

When scaling inside a node, more resource contention will exist. Figure 8 also shows the HDGEMM intra-node scalability when more CPUs and GPUs are used. When running DP on two GPU chips (DP-2GPU), the efficiency drops 11% compared to DP-1GPU. Moreover, when we extend DGEMM to two CPUs and two GPU chips system (HB-2GPU), the efficiency drops 5% from DP-2GPU. We believe the resource contention is the main reason influencing DGEMM scalability. In the following sections, we will focus on our HDGEMM intra-node scalability, and dis-

cuss system memory contention and PCIe contention in more details.

4.3.2 Scalability of Multiple GPUs

On state-of-the-art systems, most accelerators (i.e. GPU, ClearSpeed, Tilera) are attached to CPU through PCIe bus on the motherboard. Usually there are multiple PCIe slots supporting more than one GPU on one board. Besides, some GPU cards also integrate multi-chips, e.g. ATI Radeon HD5970 and NVIDIA Tesla S1070. Therefore, it is necessary to figure out how HDGEMM library scales on multiple GPUs. As for motherboards, the lane allocation among PCIe slots is different. Take two GPU boards as an example, some motherboards support x16 + x16 combination, while others only support x8 + x8. If the combination is x16 + x16, there is not PCIe contention between two GPUs boards, thus the intra-node scalability will not be influenced by PCIe bandwidth. However, if the lane is divided between two GPU boards as x8 + x8 combination, we believe the PCIe contention is the same as two chips inside one GPU board with an x16 slot. In this paper, we focus on the situation that PCIe contention exists and influences DGEMM scalability, i.e. multiple GPU boards limited by the total lane number, and multiple GPU chips inside one GPU board. Therefore it is fair to conclude the trend of scalability using multiple GPU chips. Due to the limitation of our experimental platform, we run two processes each of which is in charge of a pair of one CPU and one GPU chip. The experiments profile the alterations of the effective bandwidth from one GPU chip to two GPU chips, and are intended to analyze bandwidth contention between two GPU chips to predict its scalability with more GPUs inside a computing node. For comparison, each process of DP-2GPU runs with the same problem scale as that of DP-1GPU. From Table 2, the bandwidth contention exists on both PCIe bus and system memory.

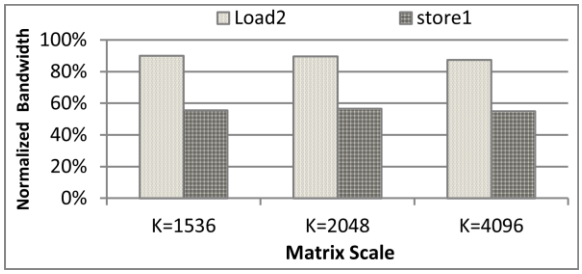

Figure 9. PCIe bandwidth achieved by *load2* and *store1* stages on a two GPU chips experiment, normalized to one GPU scenario

In order to focus on the bandwidth variations, the measured bandwidth of DP-2GPU is normalized to that of DP-1GPU. First, we consider PCIe contention during the execution of *load2* and *store1*. The reduction of average bandwidth is shown in Figure 9 with the normalized values in y-axis. As shown in this figure, *load2* and *store1* only get 89% and 56% of the PCIe bandwidth achieved in DP-1GPU, respectively. We see significant decreases in *store1*, which are caused by higher frequency of PCIe request and larger amount of data to be transferred (the size of C is larger than that of A and B). As we mentioned in Section 3.3, for full pipelining, a sub-matrix calculated by each *mult1* kernel is further divided into smaller matrices. And each *store1* operates on one smaller matrix. So, while *mult1* is running, there are several *store1* executing at the same time (4 *store1* stages execute in our implementation, corresponding to the size of sub-matrices calculated by *mult1* kernel). During *mult1* execution, the major PCIe bandwidth is used by *store1*, which reaches high occupancy even with one GPU chip. Therefore, when scaling to two GPU chips, PCIe bus contention becomes worse. However, the bandwidth of PCIe bus from CPU to GPU (*load2*) does not suffer so much compared to the case of GPU to CPU (*store1*). That is because *load2* is pipelined with *mult1* kernel among workunits so that the request on PCIe is not as frequent as *store1*. Besides, the size of matrices transferred by *load2* is (m+n)*k, while the size of matrix C transferred by *store1* is m*n. Since k is much less than n, the former puts less pressure on PCIe bus.

Secondly, in addition to PCIe bus, there also exists contention in system memory since we still need to transfer data between CPU application space and the remote memory in the CAL system. Figure 10 shows that the relative system memory bandwidth of *load1* and *store2* normalized to these on one GPU chip. We find that *load1* and *store2* drop 8% and 14% bandwidth respectively when scaling to two GPU chips. The reason is similar to that of PCIe contention in Figure 9, but the bandwidth reduction is smaller, because the frequency of *store2* is much less than that of *store1*.

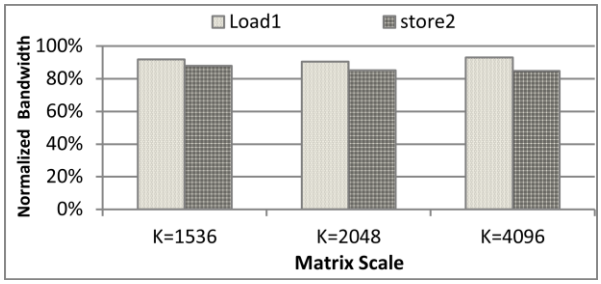

Figure 10. System memory bandwidth achieved by *load1* and *store2* stages on a two GPU chips experiment, normalized to one GPU scenario

By profiling the execution of pipeline, we find that the steps of *load1*, *load2*, *store1*, *store2* are almost overlapped completely by *mult1* kernel in the case of DP-1GPU. However, the bandwidth reduction in the two GPU case still leads to an efficiency degradation of 11% shown in Figure 8. This observation indicates that the contention on the shared resources (PCIe bus and system memory) prevents some data transfer operations from being overlapped by *mult1* kernel. As the number of GPUs increases, the bandwidth requests will become more frequent, thus the computing efficiency will decrease due to resource contention. As discussed above, when scaling from one GPU chip to two GPU chips, the achieved effective bandwidth of both PCIe and system memory is decreased. Based on the experimental results, we have the following observations:

- **Observation 1:** *Due to the contention of PCIe bus, DGEMM on multiple GPUs with limited lane hits limitations.* In DGEMM implementation, both *load2* and *store1* compete for PCIe bandwidth. As shown in Figure 7, these two phases occupy more than 60% of the total data transfer time. Our experimental results in Figure 8 show a significant decrease of bandwidth due to contention only on two GPU chips. It would be worse if more GPUs have to share the PCIe bandwidth.
- **Observation 2:** *DGEMM on multiple GPUs will not benefit much by improving system memory bandwidth.* Although both *load1* and *store2* consume system memory, it seems they are not so sensitive to the contention. Besides, their execution time is not the major part of the total execution time

(Figure 7). In very limited applications, pinned memory usage can help avoid both *load1* and *store2*. However, the assumption is needed that no data rearrangement is required, and the data fits in the limited pinned memory space. However, our work proves that algorithmic optimizations can mitigate these overhead in broader applications.

4.3.3 Scalability of Hybrid CPUs and GPUs

In our platform, Intel Xeon CPU provides a computational capability of 128GFLOP/s, which contributes 12% peak performance of the whole system. It is not negligible when trying to improve performance of compute-intensive programs like DGEMM. In our HDGEMM library, matrices are first equally split into two parts, each of which is calculated by a pair of one CPU and one GPU chip, respectively. For each pair of CPU-GPU, we adopt the partition algorithm described in [19] to assign workload between them. Although HB-2GPU improves performance by 6% on average over DP-2GPU, the efficiency in weak scalability degrades 5% compared with DP-2GPU. In this section we explain the reasons of the efficiency decrease.

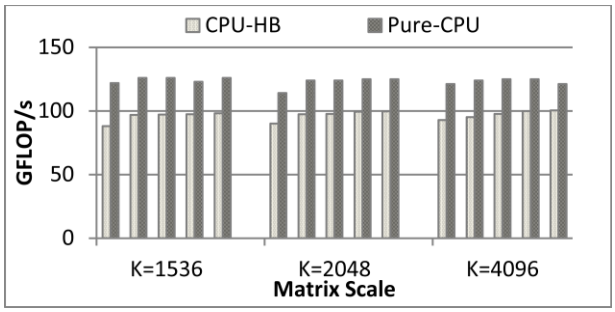

Figure 11. Performance comparison between CPU side in Hybrid DGEMM with CPU only DGEMM implememtation

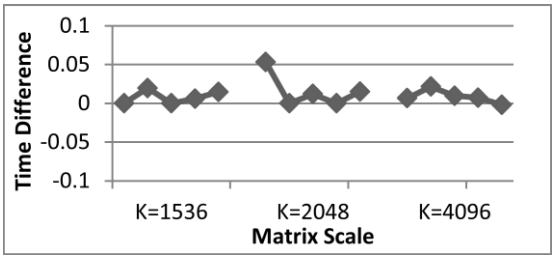

Figure 12. Influence of the task imbalance between CPU and GPU

Figure 8 also plots the performance of HDGEMM as HB-2GPU. We profile the performance contributed by CPU (denoted as CPU-HB) in Figure 11 when HB-2GPU is executed. For comparison we run a CPU-only DGEMM program (denoted as Pure-CPU), which calculates on the same matrix scale as that of CPU in HB-2GPU. This figure shows that CPU part of HDGEMM results in a performance loss of 22%. We believe that CPU-HB performance is influenced by the GPU side operations *load1* and *store2*. Data transfer to/from GPU shares the same application space with CPU DGEMM calculation. With the same reason as DP-2GPU in Section 4.3.2, we showed that system memory contention does not affect GPU part DGEMM performance in HB-2GPU. However the sharing of application space causes serious memory contention on CPU side. We further deduce another observation:

- **Observation 3:** *It will be helpful to improve HDGEMM performance by raising system memory bandwidth.* As CPU computational capacity increasing, system memory contention will make much greater influence to the whole HDGEMM library. In situations where pinned memory can be used, it could help to alleviate this contention.

Another minor reason for the efficiency degradation is the imbalanced workload partition between CPU and GPU. In our adopted partition strategy, a heuristic algorithm is used to search an appropriate split ratio of which the difference of execution time between CPU and GPU becomes less than a threshold. We take 0.1 seconds as the threshold in this paper, which is an approximate optimal value selected by multiple iterations of experiments. Figure 12 plots the execution time differences between CPU and GPU. We take CPU execution time as reference, and the differences between them are calculated by (time of GPU - time of CPU)/(time of CPU). The slight imbalance leads to a little loss of the overall HDGEMM performance. However, as the difference is small, the performance degradation caused by load imbalance is not significant (about 1%).

5. Related Work

Some previous work optimized DGEMM assuming that the matrices have already been resident in GPU on-board memory. N. Nakasato presented a new DGEMM kernel implementation on ATI HD5870 in [14], which achieved a peak performance of 87%. Our optimized DGEMM kernel gets 94% of peak performance on HD5970 with one Cypress chip. GATLAS auto-tuner [18] makes use of auto-tuning method to increase the portability among different GPU architectures and is meant to be used in real applications. GATLAS still solves DGEMM with matrices which can be resident in GPU on-board memory. Thus, there is no direct way to call GATLAS in real applications so far with large data. Satoshi Ohshima et al. optimized matrix multiplication in [25] on a CPU-GPU heterogeneous environment. They divided matrix multiplication into partial computations on CPU and GPU, and parallelized their computation using two threads without considering data transfer between them. Volkov, V., and Demmel, J. W. implemented one-sided matrix factorizations (LU, QR, etc.) on a hybrid CPU-GPU system in [11]; they divide the factorization processes to CPU and GPU separately. Matrix-matrix multiplication in that case still uses data stored in GPU on-board memory without data transfer. There are some other work in this case focusing on GPU kernel optimization without considering data transfer, we don't cite them all here.

Although there is previous work taking data transfer into account, they mostly parallelize CPU computation with GPU computation, without overlapping the data transfer process. S. Venkatasubramanian and Richard W. Vuduc implemented a hybrid Jacobi on a heterogeneous CPU-GPU architecture in [20]. They considered data transfer between CPU and GPU, and the performance improvement of hybrid implementation is only 8%. Ogata Y. et al. developed a model-based CPU-GPU heterogeneous FFT library in [24]. They proposed a performance model to better divide FFT computation between CPU and GPU. In our work we avoid load imbalance by leveraging an adaptive split algorithm in [19]. Our work focuses on solving large scale DGEMM considering data transfer between CPU and GPU on a heterogeneous architecture. We not only take data transfer overhead into account, but also optimize this process with pipelining algorithms, making data transfer overlapped by computing process. Therefore, our hybrid DGEMM can be applied in real applications. Through our optimizations, the hybrid DGEMM achieves 844GFLOP/s in maximum with 80% efficiency. Canqun Yang et

al. proposed overlapping data transfer time with DGEMM multiplication in [19], which includes load and store pipelining. We further make use of data placement approach to improve DGEMM kernel performance and develop a new pipeline. The added methods improve the overall DGEMM performance by up to 74%, which makes the greatest contribution to our optimized DGEMM. In addition, we further analyze the shared resource (especially PCIe bus and system memory) contention and the intra-node scalability of our DGEMM on multiple CPUs and GPUs. From the analysis, we give some advice for scaling DGEMM to a heterogeneous CPU-GPU architecture.

6. Conclusion

We optimized large scale DGEMM via three optimization approaches (double buffering, data reuse, and data placement) to build a new HDGEMM library on a heterogeneous architecture with CPU and ATI GPU. In this pipeline, we overlap data transfer process by DGEMM kernel execution on the GPU device. Our optimized DGEMM achieves 408 GFLOP/s and 88% efficiency on one Cypress GPU chip of ATI Radeon™ HD5970. We achieve 758GFLOP/s with 82% efficiency by running the optimized DGEMM on the whole ATI Radeon HD5970. When implemented on a heterogeneous CPU-ATI GPU system, HDGEMM reaches 80% of the peak performance, which is 844GFLOP/s. Compared with the kernel performance, we think that we have already achieved very high efficiency, and there is not much room left for further optimization on one GPU. However when scaling to multiple CPUs and GPUs intra-node, though HDGEMM performance increases progressively, the efficiency degrades with more computing units. We are convinced that the major factor is the shared resource contention, especially PCIe and system memory contention. Through the experiments and analysis, we conclude three observations to give advice for relaxing the reosurce contention in the future. 1) Due to the contention of PCIe bus, DGEMM on multiple GPUs with limited lane hits limitations. 2) The GPU part of HDGEMM on multiple GPUs will not benefit much by improving system memory bandwidth. 3) It will be helpful to improve HDGEMM performance through CPU part by raising system memory bandwidth. We consider designing a self-adaptive library in the future. It will automatically determine each level of submatrices size according to GPU registers, memory hierarchy size, system memory bandwidth and PCIe bandwidth, and then generate the most efficient pipeline.

References

[1] J. J. Dongarra, J. Du Croz, I. S. Duff, and S. Hammarling, A set of Level 3 Basic Linear Algebra Subprograms, ACM Trans. Math. Soft., 16 (1990), pp. 1--17.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK: A Portable Linear Algebra Library for High-Performance Computers, UT-CS-90-105, May 1990.

[3] HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers http://www.netlib.org/benchmark/hpl/

[4] AMD. ATI Stream SDK CAL Programming Guide v2.0, 2010.

[5] AMD Core Math Library for Graphic Processors (ACML-GPU) http://developer.amd.com/gpu/acmlgpu/pages/default.aspx

[6] AMD Accelerated Parallel Processing Math Libraries (APPML) http://developer.amd.com/libraries/appmathlibs/Pages/default.aspx

[7] NVIDIA. Compute Unified Device Architecture Programming, Guide Version 3.2

[8] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. C. Whaley and K. Yelick. Self-Adapting Linear Algebra Algorithms and Software, Proceedings of the IEEE, Volume 93, Number 2, pp 293-312, February, 2005.

[9] Goto, K., and Geijn, R. A. v. d. Anatomy of high-performance matrix multiplication. ACM Trans.Math. Softw. 34, 3 (2008), 1–25.

[10] Nath, R., Tomov, S., J. Dongarra, An Improved MAGMA GEMM for Fermi GPUs, University of Tennessee Computer Science Technical Report, UT-CS-10-655 (also LAPACK working note 227), July 29, 2010.

[11] Volkov, V., and Demmel, J. W. Benchmarking GPUs to tune dense linear algebra, 2008 ACM/IEEE Conference on Supercomputing (SC08).

[12] Ryoo, Shane and Rodrigues, Christopher I. and Baghsorkhi, Sara S. and Stone, Sam S. and Kirk, David B. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP'08), pp. 73-82, 2008.

[13] Ryoo, Shane and Rodrigues, Christopher I. and Stone, Sam S. and Baghsorkhi, Sara S. and Ueng, Sain-Zee. Program optimization space pruning for a multithreaded GPU, Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization (CGO'08), pp. 195-204, 2008

[14] N.Nakasato. A Fast GEMM Implementation On a Cypress GPU, 1st International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS 10). 2010.

[15] H.Wong, M.Papadopoulou, M. Sadooghi-Alvandi, A.Moshovos. Demystifying GPU microarchitecture through microbenchmarking, IEEE International Symposium on Performance Analysis of Systems and Software, March 2010.

[16] G.Tan, Z.Guo, M.Chen, D.Meng. Single-particle 3D Reconstruction from Cryo-Electron Microscopy Images on GPU, 23rd ACM International Conference on Supercomputing (ICS'09), 2009, pp.380-389.

[17] Li, Y., Dongarra, J., and Tomov, S. A Note on Auto-tuning GEMM for GPUs. In Proceedings of ICCS'09 (Baton Rouge, LA, USA, 2009).

[18] Jang, C. GATLAS: GPU Automatically Tuned Linear Algebra Software, http://golem5.org/gatlas/.

[19] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, Kai Lu, "Adaptive Optimization for Petascale Heterogeneous CPU/GPU Computing," cluster, pp.19-28, 2010 IEEE International Conference on Cluster Computing, 2010

[20] Sundaresan Venkatasubramanian, Richard W. Vuduc Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In Proceedings of the 23rd international conference on Supercomputing (ICS '09). ACM, New York, NY, USA, 244-255.

[21] J. Dongarra, P. Beckman, Terry Moore, et al. The International Exascale Software Project roadmap. IJHPCA 25(1): 3-60 (2011)

[22] G.Tan, N.Sun and G.R.Gao. Improving Performance of Dynamic Programming via Parallelism and Locality on Multi-core Architectures, IEEE Transactions on Parallel and Distributed Systems, Vol.20, No.2, 2009, pp. 261-274.

[23] Mark Silberstein, Assaf Schuster, and John D. Owens. Accelerating sum-product computations on hybrid CPU-GPU architectures. In Wen-mei W. Hwu, editor, GPU Computing Gems, volume 2, chapter 9. Morgan Kaufmann, August 2011

[24] Ogata, Y.; Endo, T.; Maruyama, N.; Matsuoka, S.; "An efficient, model-based CPU-GPU heterogeneous FFT library," Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on , vol., no., pp.1-10, 14-18 April 2008

[25] Satoshi Ohshima, Kenji Kise, Takahiro Katagiri, and Toshitsugu Yuba "Parallel Processing of Matrix Multiplication in a CPU and GPU Heterogeneous Environment" Proc. 7th International Conference on High Performance Computing for Computational Science (VECPAR'06), Lecture Notes in Computer Science, No. 4395, pp. 305--318, Springer-Verlag (2007)