Performance of DGEMM on a Heterogeneous Architecture with CPU and ATI GPU

Jiajia Li, Xingjian Li, Guangming Tan
Institute of Computing Technology, Chinese Academy of Sciences

Udeepta Bordoloi
Advanced Micro Devices, Inc.

ABSTRACT
In this paper we optimize double precision matrix-matrix multiplication (DGEMM) on a heterogeneous architecture with CPU and ATI GPU. Data transfer between CPU and GPU is a performance influencing factor in real applications. With respect to the effect of software pipelining on mitigating the overhead of data transfer, we develop three pipelining DGEMM algorithms, which are incrementally optimized by double buffering, data reuse and data placement. On ATI HD5970, the combined optimized DGEMM achieves 758GFLOP/s with 82% efficiency, which is more than 2 times higher than ACML-GPU v1.1. It also achieves 844GFLOP/s with 80% efficiency on a heterogeneous system with Intel Westmere EP and ATI HD5970. Further, we present a comprehensive analysis on architectural factors impacting the scalability of DGEMM on heterogeneous systems with multiple CPUs and GPUs. Our analysis shows that the contention on both PCIe and memory bus are the major sources of performance degradation 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

1. Introduction
Double-precision matrix-matrix multiplication (DGEMM) is a performance critical kernel in scientific and engineering applications. Many important numerical algorithms rely on high performance implementations of DGEMM, such as BLAS [1] and LU factorizatoin [2]. The benchmark HPL [3] for ranking supercomputers in the world implements LU factorizatoin on a dense matrix. Since its performance depends highly on the underlying hardware, all processor vendors provide highly optimized implementations of DGEMM on their own processors, e.g. Intel MKL and AMD ACML. Both Intel and AMD further optimized BLAS library on their own multi-core processors. GPU offers more than an order of magnitude speedup of peak floating-point computing power over conventional processors. For double-precision operations AMD HD5970 reaches 928GFLOP/s using two GPU chips codenamed "Cypress", and NVIDIA Tesla C2070 offers 515GFLOP/s. It is well-known that DGEMM is compute-intensive and exhibits regular memory access patterns. These features are supposed to be well suited to GPU. As a matter of fact, there are a quantity of works to accelerate DGEMM with GPUs [11-17, 19-21, 23-27]. Note that most of the previous works focus on optimizations only on GPU side when data are already resident in GPU on-board memory. Currently GPU acts as an accelerator attached to host CPU through PCIe bus. In real applicatoins, all data are initially located in CPU system memory before GPU calulates DGEMM. There are data transfers between CPU and GPU because GPU works on data located in GPU on-board memory, which is comparatively limited in size and needs to be initialized using data transferred from CPU.

In a heterogeneous system with CPU and GPU, data movement through memory hierarchy plays an important role on DGEMM performance. From a system viewpoint, there are two layers of memory hierarchy including CPU system memory and GPU local memory. It needs data transfer between CPU and GPU through PCIe bus, before GPU shader fetches data from its local memory for computation. Note that the bandwidth of GPU local memory is 256GB/s on HD5970 while PCIe only provides a peak bandwidth of 8GB/s. Although DGEMM is compute-intensive, the bandwidth gap still becomes a bottleneck for DGEMM performance on a heterogeneous system. Nakasato [15] showed that the efficiency of DGEMM decreaes from 85% to 55% when the overhead of data transfer between CPU and ATI Cypress GPU are included. A lack of DGEMM efficiency is also observed in ACML-GPU [7], which is a library released by AMD to acclerate GEMM functions on a heterogeneous system with CPU and ATI GPU.

Given the current heterogeneous CPU-GPU architecture, *how does memory hierarhcy really affect DGEMM performance*? An quantitative analysis of each layer of memory will be helpful to develop program optimization approaches and hybrid architectures in the future. Although there are several previous works to optimize DGEMM on GPUs, few available materials reported quantitative analysis. It is necessary to analyze the effect of the memory hierarchy quantitatively for summarizing some optimization techniques.

In this paper we address the above issues by inverstigating a heterogeneous system with multi-core CPU and Cypress GPU. A fast DGEMM implementation is developed on it. We resort to alternative algorithmic solutions to mitigate the overhead of data movement through memory hierarchy on current hybrid CPU-GPU architecture. Specifically, we make three main contributions in this paper:

- We quantitatively analyze DGEMM performance on a heterogeneous architecture with data transfer between CPU and GPU. The analysis identifies the undercover data transfer. The experiments find out that the time of data

transfer makes up more than 40% percentage, instead of about 20% in the previous work. This high overhead has considerable impact on DGEMM performance .

- We propose a new pipelining algorithm for large scale DGEMM through three incremental optimization approaches (Double Buffering, Data Reuse and Data Placement). Our optimized DGEMM achieves 408 GFLOP/s performance and 88% efficiency on one Cypress GPU. Compared with AMD's ACML-GPU v1.1, the optimized DGEMM is imporved by more than 2 times. The optimized DGEMM achieves 758GFLOP/s with 82% efficiency on ATI HD5970, and it also achieves 844GFLOP/s with 80% efficiency on a heterogeneous system with Intel Westmere EP and ATI HD5970.
- Finally, we analyze resource contention (especially PCIe contention and system memory contention) when scaling to multiple CPUs and GPUs. From the observations, we find that resource contention is a major bottleneck to scalability. However, it is difficult to avoid the overhead of the resource contention by a pure software solution.

2. Background

Our performance optimization focuses on ATI Evergreen GPU architecture; the high end one of that series is Cypress. HD5870 card contains one Cypress chip, and HD5970 card integrates two Cypress chips. This section highlights several important features of Cypress, especially for memory hierarchy in the ATI CAL software layer [6]. Since the DGEMM routine in ACML-GPU library is used as a baseline program for performance optimization, we profile the execution of its variant to motivate optimizations.

2.1 Cypress GPU

One Evergreen GPU chip integrates multiple compute units, a controller unit which is called ultra-threaded dispatch processor, memory controllers and DMA engines. The Cypress chip microarchitecture is featured 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 because this paper does not focus on kernel optimizations (the kernel used in this work has approached an efficiency of more than 80%). We refer to readers to [15] for the detail. Totally, it offers a peak performance of 2.32TFLOP/s in single-precision operations at core frequency of 725Mhz. Double-precision floating point operations are processed at one-fifth the single precision rate. Accordingly, it produces 464GFLOP/s in double-precision operations at 725Mhz.

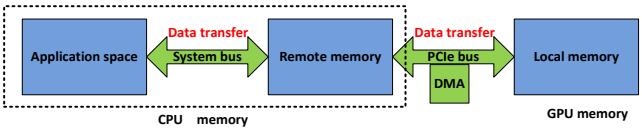

Figure 1. Memory hierarchy in CAL system between CPU and ATI GPU

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

- CAL system abstracts all physical memory into local and remote memory. Local memory corresponds to the high-speed video memory located on the graphics board. Remote memory corresponds to the memory that is not local to the given 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 in registers into remote memory by using store instructions. But such a store operation results in poor performance because access to remote memory is slower and has higher latency compared to local memory. Remote memory is partitioned into cached and uncached parts. For example, the CAL system reserves 1788MB uncached memory and 500MB cached memory on HD5970 (the main experimental platform in this paper). Therefore, *the performance is sensitive to selecting which memory region for the shared data between CPU and GPU.*
- A typical CAL application initializes data in CPU application space. There is a two-copy process: between application space and remote memory, and between remote memory and GPU device (local) memory. One optimization technique is the use of system pinned memory, where the application stores data directly into memory that can be used for DMA transfer, so that an extra copy in host memory will be skipped. It has been proven efficient to improve the performance of some applications when used in CUDA [5]. In CAL, the equivalent technique is the use of remote memory, however there are constraints such as its limited size. Therefore, *it is necessary to orchestrate data transfer son PCIe bus with proper memory management (i.e. pipelining) to reduce these bottlenecks.*

2.2 DGEMM

In this section we describe an algorithmic framework of large scale DGEMM on a heterogeneous CPU-ATI GPU system, with initial data resident on 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. In the blocking algorithm, $A=\{A_1, A_2, \ldots, A_p\}$, $B=\{B_1, B_2, \ldots, B_q\}$, $C=\{C_1, C_2, \ldots, C_{pxq}\}$, where both p and q depend on GPU memory size. For example, we assume that $p=2$ and $q=2$ for simplicity of presentation, the matrix multiplication is illustrated in Figure 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 into 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 multiply by *beta*)
7. **endfor**

Algorithm 1. Initial DGEMM implementation

The partition results in four independent sub-matrices of C which are calculated in parallel: $C_1=A_1*B_1$, $C_2=A_1*B_2$, $C_3=A_2*B_1$, $C_4=A_2*B_2$. For each sub-matrix multiplication we load its dependent sub-matrices A and B into GPU memory, then DGEMM kernel may further partition the sub-matrices into smaller ones for multiplication [9-13] (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, data can be shared between the application space and GPU via the remote memory (Line 1). In the pseudo-code of Algorithm 1, there are two processes for loading data to GPU memory (*load1*, *load2*) and one process for storing data back to CPU memory (*store*). In fact, line 5 representing DGEMM kernel execution implicitly includes another store operation that stores the resulting sub-matrices of C from registers into remote memory.

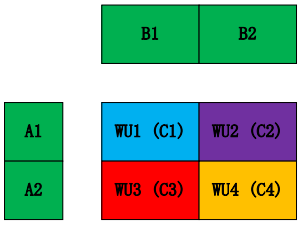

Figure 2. Workunits split

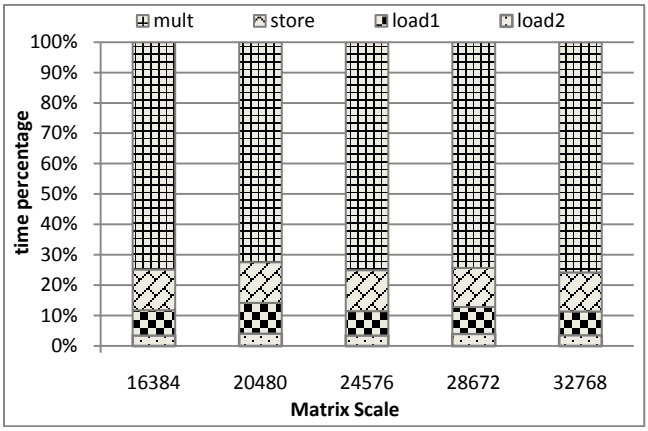

Figure 3. Time composition in initial DGEMM implementation

Table 1. Resource allocation in each step of Algorithm 1.

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

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 percentage of all problem scales shows a similar distribution, we take k=2048 for example, and give the matrix scale of m (n) in x-axis. We learn that *mult* (kernel) occupies the most percentage (more than 70%) while the three data transfer steps are summed to be less than 30%. Intuitively, the overhead of data transfer may be hidden by overlapping them with the kernel execution. In order to find a way to achieve this overlap, we classify the resources used by the algorithm into GPU, CPU+Memory bus and PCIe bus. The operations on these three resources can proceed in parallel. Table 1 outlines the resource allocation in each step of Algorithm 1. Both *load1* and *store* consume the resources of CPU+memory bus, and *load2* only needs PCIe bus to transfer data. The *mult* kernel is executed on GPU, and then outputs its results to CPU through PCIe bus. Based on the observation of resource allocation in each step, it is straightforward to implement a software pipelining algorithm to overlap the data transfer (*load1*, *load2*, *store*) with the kernel *mult*. In the previous works, ACML-GPU overlaps *some stages with* a *mult* stage. Further, Yang et.al implemented pipelining algorithm that overlap *load1* with *mult*. However, as shown in their literature and our experiments in Section 4, the simple pipelining algorithm improves performance a little (about 20%). In fact, the most execution time of Algorithm 1 is occupied by the *mult* kernel. The kernel directly dumps results from registers to remote memory in the previous algorithms. In addition to the computation on GPU, the kernel is involved with a data transfer through PCIe bus. The data transfer included in the kernel becomes a performance bottleneck because: i) PCIe bandwidth is much less than memory bandwidth of either CPU or GPU, ii) the size of transferred data in *mult* may be larger than that in *load2*. For the example of DGEMM in LINPACK benchmark, *k* is much less than both *m* and *n*, thus the size of resulting matrices is *m*n* that is larger than *k*(m+n)*. Besides, as described in the next section we can reduce the times of data transfer in *load2* by exploiting data reuse. Therefore, we refine the pipelining algorithm to achieve better overlap between floating-point operations and data transfer operations.

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 memory operations: *load1*, *load2*, *store*.

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 C is partitioned into blocks*
//
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 into 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 multiply by *beta*)
      //mult
8. calculate $C_{i,j}$ on GPU device and output it to remote memory
9. ***endfor***
      //store
10. copy $C_{i,j}$ form remote memory to application space (also multiply by *beta*)
11. **endfor**

Algorithm 2. DGEMM with Double Buffering

3.1 Double Buffering

The *store* for writing back C matrix performs both data transfer and a small portion of floating-point calculations by consuming CPU resources (*beta * C*). 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 start up when workunit *i* is executing the *store*. There are two problems in such a pipeline. First, the *load1* has a resource conflict with the *store* because both of them are performing data transfer between application space and CAL remote memory. The resource conflict degrades efficiency of the pipeline. Second, the size of remote memory is limited, especially for the cached remote memory. Since the *store* is executed at CPU side, it's better to use the cached remote memory to hold the resulting matrices of C. 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 in the same workunit, instead of between multiple workunits. Algorithm 2 shows the fine-grained pipelining execution. The algorithm further partitions the sub-matrices of C into many blocks, which are computed in a pipelined way. In order to implement the pipeline, a double buffering algorithm is adopted. We allocate two buffers in the cached remote memory. For each loop *j* in line 6-9 of Algorithm 2, the *store* dumps one buffer with a block $C_{i,j}$ to application space when the *mult* is calculating the next $C_{i,j+1}$ and filling it into the other buffer. During the process of pipelining, the two buffers are used by the *mult* and *store* in an interleaved manner. Since *mult* kernel on GPU device is executed as a non-blocking call, it proceeds in parallel with the *store* for each loop.

Partition: A={A_1, A_2, …,A_p}, B={B_1,B_2,…,B_q}, C={C_1, C_2, …,C_{pxq}}
Work unit: WU={$C_1=A_1*B_1$, $C_2=A_1*B_2$,...}
C_{ij}: the sub-matrices C is partitioned into blocks
//
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 into 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 multiply by *beta*)
//mult
8. calculate $C_{i,j}$ on GPU device and output it to remote memory
9. ***endfor***
//store
10. copy $C_{i,j}$ form remote memory to application space (also multiply by *beta*)
11. ***endfor***

Algorithm 3. DGEMM with Data Reuse

3.2 Data Reuse

Since the sum of the *load1, load2, store* execution time is much less than that of the *mult* (Figure 3), their overhead is easily hidden by the pipelining execution at the level of workunits. However, there exist resource conflicts to CPU memory bandwidth between the *load1* and the *store*, and conflicts to PCIe between the *load2* and the *mult*. The resource conflicts lead to delay in the pipeline and degrade performance.

Fortunately, we can exploit data reuse between two consecutive workunits. Take the example in Figure 2, if we schedule the execution of workunits with an order of $WU_1=\{C_1=A_1*B_1\}$, $WU_2=\{C_2=A_1*B_2\}$, $WU_3=\{C_4=A_2*B_2\}$, $WU_4=\{C_3=A_2*B_1\}$, every two consecutive workunits reuse one of the two input matrices. In order to exploit such data reuse to mitigate 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 to 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, which making the top (or bottom) matrix block in one bar connecting with the top (or bottom) block in the next bar. Second, two indicators are set to be addresses of matrices A and B in the current workunits, so that the next workunit avoids loading the same matrix blocks again. Algorithm 3 outlines the pipelining algorithm of the workunits with data reuse.

Partition: A={A_1, A_2, …,A_p}, B={B_1,B_2,…,B_q}, C={C_1, C_2, …,C_{pxq}}
Work unit: WU={$C_1=A_1*B_1$, $C_2=A_1*B_2$,...}
C_{ij}: the sub-matrices C is partitioned into blocks
//
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 into 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 multiply by *beta*)
//mult
8. ***DMAPipeline***($C_{i,j}$)
9. ***endfor***
//store2
10. copy $C_{i,j}$ form remote memory to application space (also multiply by *beta*)
11. ***endfor***

Algorithm: DMAPipeline($C_{i,j}$)
$C_{i,j,k}$: the $C_{i,j}$ blocks are further partitioned into sub-blocks
//
//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 into remote memory
//mult1
4. calculate $C_{i,j,k}$ in local memory
5. ***endfor***
//store1
6. DMA transfer $C_{i,j,k}$ from local memory into remote memory

Algorithm 4. DGEMM with Data Placement

3.3 Data Placement

In order to make use of the data reuse, the matrices of 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 executed by CPU; (ii) the cached remote memory is too small to effectively accelerate *load1*. Since matrices of C in all workunits are independent, there is no

data reuse for the matrices. It seems to be reasonable that the matrices of C are stored in remote memory instead of GPU local memory, because the local memory is very limited on GPU device. Therefore, in the above three algorithms the matrices of C is stored in cached remote memory for better CPU performance (i.e. memory copy and multiply by *beta*).

Thanks to the pipelining execution, memory transfer operations (*load1*, *load2*, *store*) are overlapped with the kernel multiplications (*mult*). The execution time of the pipeline is determined by the *mult* in the pipelined algorithm. Therefore, one direction of the remaining optimizations is to reduce the length of *mult* in the pipeline. Note that the *mult* kernel contains memory operations which directly dump the values of 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 be overlapped with GPU kernel computation, too.

In order to construct a new phase in the pipeline, the memory operations for outputting matrices of C into remote memory are separated from the *mult* kernel. The kernel output is changed so that the resulting matrices of C are written into GPU local memory. Thus, as shown in Algorithm 4, the original *mult* is further split into two phases: *mult1* and *store1*. The *store1* transfers data from local memory to remote memory. With respect to resource occupancy, the *mutl1* is executed by GPU device while the *store1* is finished by DMA engine. Due to the asynchronous execution of both kernel and DMA operations, the two operations can be executed in parallel by adopting the double buffering strategy in local memory for the resulting C matrices. For more clarity, we will refer to the *store* operation as *store2* in the following sections.

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

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

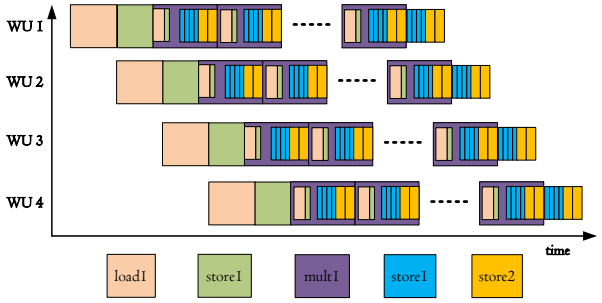

Figure 4. Our optimized DGEMM pipeline sketch

So far, we have a five stage (*load1*, *load2*, *mult1*, *store1*, and *store2*) pipeline in the optimized DGEMM. In Table 2, we depict the resource allocation in our optimized DGEMM. In this way, we have managed to solve the two problems proposed in ACML-GPU mentioned in Section 2. The *mult1* kernel no longer needs PCIe bus and system memory, except for GPU device. Therefore, without PCIe conflicts, *mult1* kernel can execute in parallel with *load2*. Algorithm 4 not only allows for a fast kernel implementation, but also makes the new pipeline much finer and alleviates delay caused by resource conflicts. We further depict the pipeline time-space diagram in Figure 4. The figure is just a schematic sketch, which doesn't consider every mini-step resource conflicts between different workunits. We use different colors to show the five steps. The bars inside a *mult1* block represent the data transfer of these sub-matrices, which are overlapped by the *mult1* kernel. As shown in the figure, except for prologue and epilogue of the pipeline, most of data transfer in Algorithm 4 can be fully overlapped.

4. Experiment Results and Analysis

4.1 Experimental Setup

The experiments are performed on a heterogeneous system composed of two Intel Xeon 5650 CPUs and one ATI HD5970 GPU card. Table 3 summarizes the configuration parameters of the experimental platform. The multi-core CPUs provides a peak double-precision performance of 128GFLOP/s. The CPU memory subsystem 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 to 256GB/s. The total peak performance of the heterogeneous system is 1056GFLOP/s.

Table 3. System configuration of the experimental platform.

processors		Intel Xeon X5650	ATI 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

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

- *DB*: The program implements Algorithm 2. It uses a double buffering strategy to hide the overhead caused by writing the resultant C matrices from remote memory to application space. The program allocates two buffers, 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 for overlapping the operations of loading the input matrices. An important optimization to the pipelining algorithm is to exploit data reuse of 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 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 both CPU and GPU perform multiplications cooperatively. The matrices are partitioned between CPU and GPU. We adopt the strategy proposed in [21] for workload balance between them. In our experiments, we startup two MPI processes, each of which uses a pair of one CPU and one GPU.

Note the four programs represent four incremental optimization strategies. Each program adds another optimization strategy by following the order of DB<DR<DP<HB. The source of the ACML-GPU [7] library that implements DGEMM on the GPU is available with the package. We use the code for our initial experiments and for comparing and evaluating the described optimizations in this paper.

Table 4. The sizes of *m,n,k* in the experiments.

k	*m=n*				
1536	16384	20480	24576	28672	32768
2048					
4096					

Table 4 shows the scales of the matrices (*m,n,k*) in the experiments. The sizes of *m* equals to those of *n* because the difference between these two values has little effect on performance. The size of *k* determines the amount of data reuse during reading the input matrices and 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 of all the three k sizes by calculating the average performance in the same size of k. 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

First, we report the overall performance of our optimized DGEMM on a heterogeneous system. The baseline program for performance comparison is the ACML-GPU library version 1.1. Figure 5 plots the performance improvements by optimizing the pipelining algorithms (DP) and assigning parts of workload to CPU (HB). Our hybrid DGEMM (HB-2GPU) achieves a maximal performance of 844GFLOP/s and an efficiency of 80% for *(m,n,k) = (16384, 16384, 4096).* The optimized DGEMM on GPU (DP-2GPU) reaches a maximal performance of 758GFLOP/s and an efficiency of 82% for *(m,n,k) = (16384, 16384, 4096).* These results show that DP-2GPU improves DGEMM's performance by 2.0 times on average over the ACML-GPU library. The hybrid program further improves performance by 10%-20%. Basically, all of the three programs show an increase in both performance and efficiency with larger matrices. There are few cases with abnormal performance (e.g. m=m=10240) because the problem scales are not multiples of the optimal blocking size for data transfer and kernel execution. As the matrix scales become larger, the performance improvement of DP over ACML-GPU drops. For k is 1536, 2048, 4096, the speedup is 2.9X, 2.1X, and 1.9X respectively. This is expected because the ratio of data transfer to kernel execution decreases with larger problem scales, and the effect of our pipelining optimizations is to mitigate the overhead of data transfer between CPU and GPU. We would like to point out that achieving high efficiency on GPUs for large datasets is relatively easier when compared to small datasets. Because in the latter case, data transfer makes up a large overhead. It is the relative small datasets that our optimizations show their efficacy. As shown in the figure, we also know HB-2GPU has more improvement with larger matrices size. That's because CPU gains better performance, and enhances the whole performance of HB-2GPU, when matrices size becomes larger.

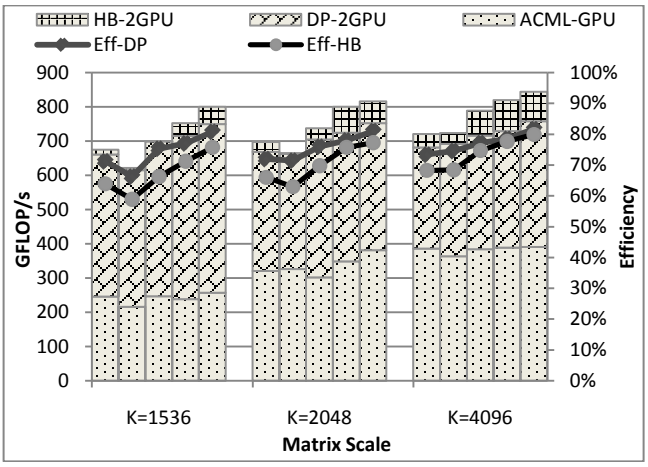

Figure 5. Our optimized DGEMM Performance and efficiency with two MPI processes

Second, we evaluate the effect of the three optimization strategies described in Section 3. In order to isolate other noises (i.e. 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, who only performs data transfer cooperatively. Figure 6 plots the increments in GFLOP/s achieved by the optimized algorithms of double buffering (DB), data reuse (DR) and data placement (DP), respectively. Compared to Algorithm 1, double buffering improves 16% performance by pipelining *store2* inside a workunit. Then, data reuse further improves performance by 18%, with load operations overlapped by the multiplication between different workunits. Finally, the performance is significantly improved by 74% using data placement optimization, 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 performance with 88% efficiency on one Cypress GPU.

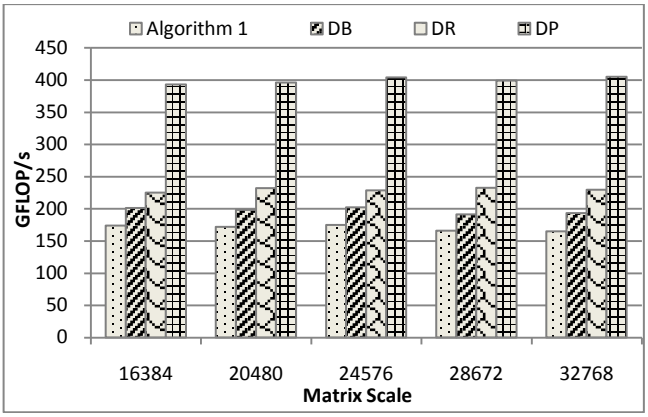

Figure 6. Performance improvement of each optimization approach

Figure 3 shows that the three data transfer steps (*load1*, *load2*, and *store*) just occupy nearly 30% of the total execution time, as its statistics does not include the time of *store1*. Our optimization

refines the pipelining algorithm and separates *store1* from the *mult* kernel. Obviously, our approach is more close to the nature of the pipelining design. In fact, counted the data transfer in *mult* kernel, the total data transfer occupies more than 40% of the total execution time observed in the experiments. We further give the time percentage of each data transfer step (*load1*, *load2*, *store1*, and *store2*) in Figure 7, which is the division of the execution time of each step to the total data transfer time. From this figure, the performance improvement by each optimization in Figure 6 is in accordance with the data transfer time distribution. It is demonstrated that our optimizations make full use of pipelining. Besides, there is an additional performance improvement to data placement because our implementation optimizes the kernel a little bit. We also learn from Figure 6 that our optimized DGEMM performance is quite steady for all the matrices sizes. This property lays a good foundation for scaling our optimized DGEMM to multiple CPUs and GPUs. The performance 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.

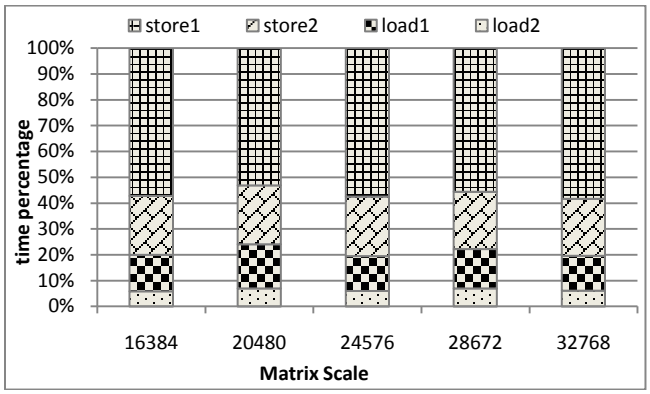

Figure 7. Percentage of the four data transfer steps in optimized DGEMM

4.3 Analysis

It is common that the DGEMM of CPU's math library can achieve a maximal efficiency of more than 90%. The hybrid DGEMM on the heterogeneous system achieves the maximal efficiency of 82%. In this section we investigate that (i) how much optimization room is left beyond our pipelining optimization on such a heterogeneous architecture; (ii) how the architectural factors affect our optimized DGEMM performance on 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 [15] optimized DGEMM kernel performance and reported the highest efficiency of 87% on HD5870 with 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 the higher efficiency of 94% on one Cypress chip.

Figure 8 plots the efficiency comparison among the kernel, DP on one GPU chip, DP on two GPU chips, and hybrid DGEMM on both two CPUs and GPUs. For each test set, we give their average efficiency. The execution of DGEMM calls the kernel for multiple times. We measure the kernel's performance in GFLOP/s for each time and calculate their average values as its final GFLOP/s in the figure. Our optimized kernel only reads and writes GPU's local memory. Its performance has no relation to the system resources at CPU side. As shown in the figure the kernel's efficiency is more than 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 the data transfer between CPU and GPU through system memory bus and PCIe bus. The optimized programs proposed in this paper use software pipelining to overlap the overhead of data transfer. The experimental results in this figure show that DP-1GPU degrades 6% performance from kernel due to the data transfer. There are two reasons for the performance degradation. First, the prologue and epilogue of the pipeline cannot be hidden, which occupy around 3% of the total DGEMM execution time. Second, as shown in Table 2, there are still intrinsic resource conflicts during the pipeline. During the pipelined execution, there may be resource conflicts to PCIe bus between *load2* and *store1*, and system bus between *load1* and *store2*. Apart from the two factors, we consider DP-1GPU achieves nearly the best performance on one GPU chip.

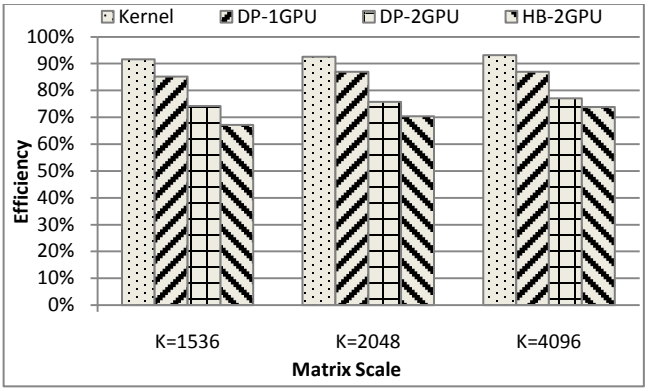

Figure 8. The efficiency of our optimized DGEMM when scaling to multiple CPUs and GPUs

Besides, from Figure 8, we also learn that the efficiency decreased when more GPUs and CPUs are used in DP-2GPU and HB-2GPU. When running DP on two GPU chips (DP-2GPU), the efficiency drops 11% compared to DP-1GPU. That's because DGEMM on two GPUs both need data transfer from CPU through system memory bus and PCIe bus. In this case, there exists contention in both the two buses. Moreover, when we extend DGEMM to two CPUs and two GPU chips system (HB-2GPU), the efficiency drops 5% from DP-2GPU. For HB-2GPU, CPU performs part of DGEMM cooperating with GPU on the shared matrices, increasing system memory bus burden. The added system memory contention decreases HB-2GPU performance from DP-2GPU. We will discuss the two types of contention (system memory contention and PCIe contention) in more detail in the following sections.

4.3.2 Scalability of Multiple GPUs

On state-of-the-art systems, most of 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, there exist multi-chips in some GPU cards, e.g. ATI Radeon HD5970 and NVIDIA Tesla S1070. Therefore, it is necessary to figure out how the optimized DGEMM algorithm scales on multiple GPUs. Due to the limitation of our experimental platform, we run two MPI processes, each of which in charge of one GPU chip. We argue that it is fair to conclude the trend of scalability using multiple GPUs because the critical factor on scalability is the shared

resource like PCIe and memory bandwidth. Our experiments intend to analyze bandwidth contention between two GPU chips to predict its scalability with more ones. The experiments profile the alterations of the effective bandwidth from one GPU chip to two GPU chips. For comparison, each MPI process of DP-2GPU runs the same problem scale as that of DP-1GPU. From Table 2 above, the bandwidth contention occurs on both PCIe and memory bus.

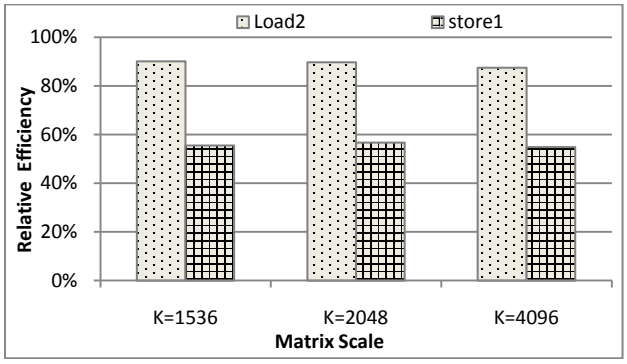

Figure 9. PCIe relative bandwidth efficiency on two GPU chips compared with one GPU chip

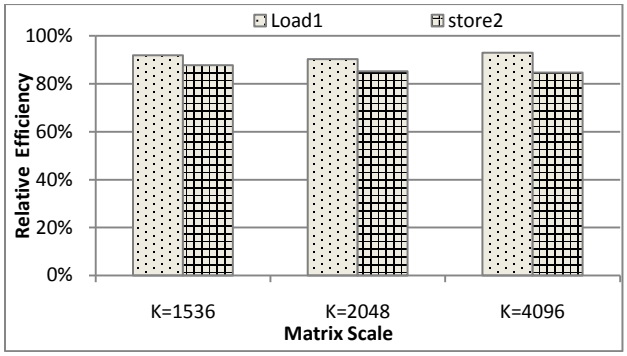

Figure 10. System memory relative bandwidth efficiency on two GPU chips compared with one GPU chip

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* 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 also an increased amount of data to be transferred (size of C is larger than 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, when *mult1* is running, there are several *store1* executing at the same time (4 *store1* executing in our implementation due to the sub-matrices size calculated by kernel). During the *mult1* execution time, the major PCIe bandwidth is considered to be used by *store1*, which reaches high occupancy even on 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*) is not suffered so much as from 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, *load2* transfers the matrices of size *(m+n)*k* while *store1* transfers the matrices of size m*n. Since *k* is much less than *n*, the latter puts more pressure on PCIe bandwidth.

Second, in addition to PCIe bandwidth, there also exists contention in memory bus because we still need to transfer data between CPU application space and the remote memory in the TI CAL system. Figure 10 shows that the memory bandwidth in *load1* and *store2* drop 8% and 14%, respectively. Due to the same reasons as with the PCIe contention example above, there is more reduction in *store2*.

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

- **Observation 1:** *DGEMM scales limitations for multiple GPUs on the same board due to sharing PCIe bandwidth.* In DGEMM implementation, both *load2* and *store1* consume PCIe bandwidth. As shown in Figure 7, these two phases occupy more than 60% of the total data transfer time. Our experiments result in Figure 8 shows a significant reduction of bandwidth due to contention between only two GPU chips. It would be worse if more GPUs have to share the PCIe bandwidth.
- **Observation 2:** *GPU-only DGEMM (both DP-1GPU and DP-2GPU) will only benefit a little bit by improving system memory bandwidth.* Although both *load1* and *store2* consume memory bandwidth, it appears to be 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 use cases, pinned memory usage can help avoid both *load1* and *store2*. However, that assumes no data rearrangement is required, and the data fits within the limited pinned memory space which is a rare resource. However, our work proves that these overhead can be mitigated by algorithmic optimizations also.

4.3.3 Scalability of Hybrid CPUs and GPUs

In our testbed, 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 hybrid DGEMM implementation HB-2GPU, matrices are first equally split into two parts, each of which is calculated by a pair of one CPU and one GPU, respectively. For each pair of CPU-GPU, we adopt the partition algorithm described in [21] to assign workload between them. Although HB-2GPU improves performance by 6% on average over DP-2GPU, the efficiency degrades 5% compared with DP-2GPU. In this section we explain the reasons of the efficiency decrease.

Figure 8 also plots the performance of DGEMM executed on hybrid CPUs-GPUs. We profile the GFLOP/s 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 the same matrix

multiplication as that of CPU in HB-2GPU. This figure shows that the hybrid program results in a performance loss of 22% for DGEMM on CPU only. We believe that it is caused by extra data copy from application space to CAL remote memory space. In last section, we showed that system memory contention does not affect DGEMM performance of GPU part in HB-2GPU, with the same reason as DP-2GPU. However the data copy incurs more memory contention on CPU side. We further educe another observation:

- **Observation 3:** *It will be helpful to decrease system memory contention and improve hybrid CPUs-GPUs DGEMM performance by raising system memory bandwidth.* As CPU computational capacity increasing, system memory contention will make much deeper influence to the whole hybrid DGEMM implementation. In situations where pinned memory can be used, that could help alleviate this contention.

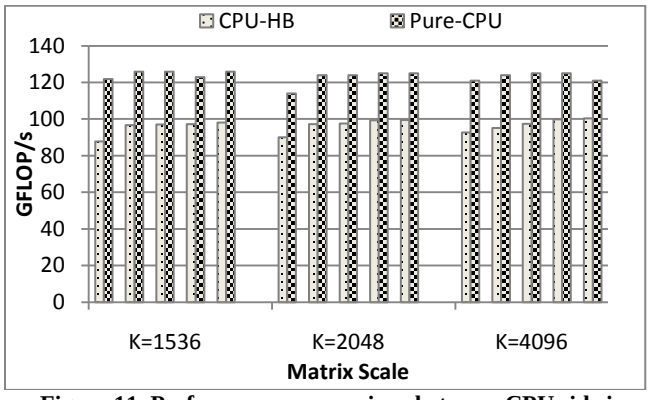

Figure 11. Performance comparison between CPU side in Hybrid DGEMM with CPU only DGEMM implememtation

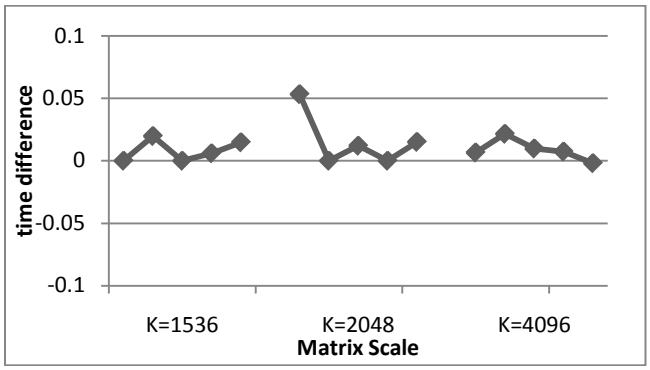

Figure 12. Influence of the task imbalance between CPU and GPU

Another minor reason for the efficiency degradation is load imbalance when partitioning workload between CPU and GPU. In our adopted partition strategy, a heuristic algorithm is used to search an appropriate split ratio that makes the difference of execution time between CPU and GPU 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 $\frac{time\ of\ GPU - time\ of\ CPU}{time\ of\ CPU}$. The slight imbalance leads to a little loss of the overall hybrid DGEMM performance. However, as the difference is small, the performance degradation caused by load imbalance is not significant (about 1%).

5. Related Works

Some works optimized DGEMM with the matrices which have already been resident in GPU on-board memory. N.Nakasato presented a new DGEMM kernel implementation on ATI HD5870 in [15], which achieved 87% peak performance. Our optimized DGEMM kernel gets 94% of peak performance on HD5970 with one Cypress chip. GATLAS auto-tuner [20] 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 scale. Volkov, V., and Demmel, J. W implemented one-sided matrix factorizations (LU, QR, etc.) on a hybrid CPU-GPU system in [12], they divide factorization processes to CPU and GPU separately. Matrix-matrix multiplication in the case still uses data stored in GPU on-board memory without data transfer. There are some other works in this case focusing on GPU kernel optimization without considering data transfer, we don't name them all here for short.

Although there are some works taking data transfer into account, they mostly parallel 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 [23]. They considered data transfer between CPU and GPU, and the performance improvement of hybrid implementation is merely 8%. DaQi Ren, Reiji Suda implemented Large Matrices Multiplication on a heterogeneous platform with multi-core CPU and GPU in [25]. They focused on power efficiency, and utilized multithreading on CPU side. CPU will switch to a new thread handling another workload, while another thread waits for GPU polling memory responses signals. Through this method GPU computation can run in parallel with CPU execution, however data transfer process still stalls the whole execution time and wastes CPU and GPU computational capacity. C. Feichtinger et al. implemented a parallel Lattice Boltzmann approach for Heterogeneous CPU–GPU Clusters in [24]. They minimize the data transfer amount by only transferring boundary values of the PDFs. For the mere performance increase of their heterogeneous implementation, they believe that load imbalance is one reason. Ogata Y. et al. developed a model-based CPU-GPU heterogeneous FFT library in [27]. They proposed the performance model to better divide FFT computation between CPU and GPU. While in our work we avoid load imbalance by leveraging an adaptive split algorithm in [21]. 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 [21], 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 scalability of our DGEMM on multiple CPUs and GPUs. From the analysis, we give some advice for scaling DGEMM to a heterogeneous CPUs and GPUs architecture.

6. Conclusion

We optimized large scale DGEMM via three optimization approaches (double buffering, data reuse, and data placement) to generate a new pipelining algorithm. In this pipeline, we overlap data transfer process by DGEMM kernel execution on GPU device. Our optimized DGEMM achieves 408 GFLOP/s performance and 88% efficiency on one Cypress GPU chip of ATI HD5970. We achieve 758GFLOP/s with 82% efficiency by running the optimized DGEMM on ATI HD5970. When implemented on a heterogeneous CPU-ATI GPU system, the hybrid DGEMM performance reaches 80% of the peak performance, which is 844GFLOP/s in maximum. With the comparison with the kernel performance, it is considered that we have already achieved very high efficiency, and there is not much room left for further optimization. However when scaling DGEMM to multiple CPUs and GPUs, there still exist factors influencing the scalability of DGEMM. The main factor is the shared resource contention, especially PCIe and system memory contention. Through the experiments and analysis, we conclude three observations. 1) Due to sharing PCIe bandwidth, DGEMM scales limitations for multiple GPUs on the same board. 2) DGEMM implementation on multiple GPUs will only benefit a little bit by improving system memory bandwidth. 3) It will be helpful to improve the performance of hybrid DGEMM on multiple CPUs and GPUs by raising system memory bandwidth.

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

[5] D. Kirk and W. W. Hwu. *ECE 489AL Lectures 8-9: The CUDA Hardware Model*, http://courses.ece.illinois.edu/ece498/al/Archive /Spring2007/lectures/lecture8-9-hardware.ppt, 2007.

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

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

[8] NVIDIA. *CUDA Community Showcase*. http://www.NVIDIA.com/ object/ cuda_apps_flash_new.html.

[9] 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.

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

[11] Nath, R., Tomov, S., Dongarra, J. *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.

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

[13] 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.

[14] 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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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).

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

[21] 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

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

[23] 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.

[24] Christian Feichtinger, Johannes Habich, Harald Köstler, Georg Hager, Ulrich Rüde, Gerhard Wellein. A Flexible Patch-Based Lattice Boltzmann Parallelization Approach for Heterogeneous GPU-CPU Clusters. CoRR, 2010

[25] DaQi Ren, Reiji Suda, "Power Efficient Large Matrices Multiplication by Load Scheduling on Multi-core and GPU Platform with CUDA," cse, vol. 1, pp.424-429, 2009 International Conference on Computational Science and Engineering, 2009

[26] 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 To appear

[27] 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