An Optimized Large-Scale Hybrid DGEMM Design for CPUs and ATI GPUs

Jiajia Li, Xingjian Li, Guangming Tan, Mingyu Chen, Ninghui Sun
State Key Laboratory of Computer Architecture
Institute of Computing Technology, Chinese Academy of Sciences
Beijing, China
{lijiajia,lixingjian,tgm,cmy,snh}@ict.ac.cn

ABSTRACT

In heterogeneous systems that include CPUs and GPUs, the data transfers between these components play a critical role in determining the performance of applications. Software pipelining is a common approach to mitigate the overheads of those transfers. In this paper we investigate advanced software-pipelining optimizations for the double-precision general matrix multiplication (DGEMM) algorithm running on a heterogeneous system that includes ATI GPUs. Our approach decomposes the DGEMM workload to a finer detail and hides the latency of CPU-GPU data transfers to a higher degree than previous approaches in literature. We implement our approach in a five-stage software pipelined DGEMM and analyze its performance on a platform including x86 multi-core CPUs and an ATI $Radeon^{TM}$ HD5970 GPU that has two Cypress GPU chips on board. Our implementation delivers 758 GFLOPS (82% floating-point efficiency) when it uses only the GPU, and 844 GFLOPS (80% efficiency) when it distributes the workload on both CPU and GPU. We analyze the performance of our optimized DGEMM as the number of GPU chips employed grows from one to two, and the results show that resource contention on the PCIe bus and on the host memory are limiting factors.

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 (SIMD)

General Terms

Algorithms, Performance, Experimentation

Keywords

High Performance Computing, Heterogeneous Architecture, GPU, DGEMM

ICS'12, June 25–29, 2012, San Servolo Island, Venice, Italy.

1. INTRODUCTION

Double-precision GEneral Matrix Multiplication (DGEMM) is a performance critical kernel found in many scientific and engineering applications. Since its performance depends on the underlying hardware, hardware vendors often provide optimized implementations of DGEMM within the BLAS library [8] (e.g. Intel MKL and AMD ACML). With the increasing popularity of GPUs and the high performance they make available, the need for GPU-optimized DGEMM implementations arise as well. DGEMM is compute-intensive and exhibits regular memory access patterns; these properties make it well suited to GPUs. Plenty of work in literature [7, 11–13, 16–23] has presented GPU-accelerated DGEMM implementations, but this work mostly focuses on optimizing computation and assumes that data can stay resident in GPU memory at all times. Unfortunately, realistic DGEMM applications exhibit operands that won't fit in GPU memory: this makes CPU-GPU data transfers crucial for the feasibility and the performance of the applications.

Optimizing CPU-GPU data transfers is non-trivial because these transfers occur through the PCIe bus, that acts as a bottleneck: while the GPU memory can deliver a throughput of up to 256 GB/s on a HD5970 GPU, a PCIe 2.0 bus only provides a peak bandwidth of 8 GB/s in each direction. Nakasato [13] reported that the floating-point efficiency of his DGEMM implementation decreased from 87% to 55% once he counted data-transfer overheads. A similar phenomenon occurs in the ACML-GPU library [2] released by AMD to accelerate BLAS functions on heterogeneous CPU-ATI GPU systems.

Software pipelining has been traditionally employed to mitigate the latency of data transfers through overlapping computation and transfers. Yang et al. [23] have developed a four-stage pipelined DGEMM algorithm that improves performance by about 30%. Automated approaches like the one proposed by Jablin et al. [10] simplifies the parallelization of workloads and handles CPU-GPU transfers automatically, but their performance still does not compare favorably with manual implementations. Therefore, the need is still strong to explore the full potential of manual data-transfer optimizations in critical kernels like DGEMM.

Multiple GPU boards can be employed on a single node in an attempt to achieve higher performance. We analyze the behavior of our optimized DGEMM on a multi-GPU node. The analysis shows that the performance increase comes at the cost of a lower floating-point efficiency (Section 4), even if there's no data dependence among tasks running on the different GPUs. We conclude that the efficiency loss is a

result of resource contention on both the PCIe bus and host memory. We believe that our conclusions should guide the configuration and dimensioning of future CPU-GPU systems intended to tackle workloads similar to DGEMM with a high degree of efficiency.

In this paper we examine performance tuning of DGEMM on a heterogeneous system comprising x86 multi-core CPUs and ATI Cypress GPU chips. We develop a new software-pipelined design of the DGEMM kernel by identifying design choices that are performance-critical on the particular ATI GPU architecture, specifically the choices of memory addressing modes and the data placement in memory.

The main contributions of this paper are:

- We develop a new software pipelining design for the DGEMM kernel running on CPU-GPU heterogeneous architectures. The new design reduces the overheads of data movement by using the image addressing mode with low latency for dumping registers to GPU memory and employing fine-grained pipelining to hide CPU-GPU data transfer latencies. We generalize our five-stage design so that it can be employed in a general heterogeneous programming framework.
- We provide an optimized implementation of our design and evaluate its performance. Experimental results show that our implementation achieves 408 GFLOPS (88% efficiency) on one Cypress GPU chip, 758 GFLOPS (82% efficiency) on two Cypress GPU chips, and 844 GFLOPS (80% efficiency) on a system comprising two Intel Westmere-EP CPUs and one ATI $Radeon^{TM}$ HD5970 GPU. Compared with AMD's ACML-GPU v1.1.2, our DGEMM implementation improves performance by more than 2 times.
- We report a comprehensive experimental analysis showing that the major scaling bottleneck when multiple GPU chips are used is resource contention, especially PCIe contention and host memory contention. We show that it is difficult to further decrease the overheads by software means. We present recommendations on how to relax the resource contention in future hardware designs, that are beneficial on any heterogeneous architectures that integrate multiple CPUs and PCIe-attached accelerators.

The rest of the paper is organized as follows. In Section 2, we quantitatively analyze previous DGEMM implementation on the heterogeneous architecture. Our new software-pipelining algorithm is proposed in Section 3. Section 4 gives the performance results and the detailed analysis. Related work is presented in Section 5, and the conclusions in Section 6.

2. BACKGROUND AND MOTIVATION

2.1 The ATI GPU Architecture

The architecture we target in this work is a heterogeneous system based on ATI $Radeon^{TM}$ HD5970 GPU, which integrates two Cypress chips into one card. The Cypress chip contains several hundreds of computing units, a controller unit named ultra-threaded dispatch processor, memory controllers and DMA engines. The microarchitecture is customized with single-instruction-multiple-data (SIMD) and very long instruction word (VLIW) for high throughput of floating-point operations. It offers a peak performance of 464 GFLOPS with double-precision floating-point operations at 725 MHz.

A notable feature related to software-pipelined design is its memory hierarchy exposed in ATI Compute Abstraction Layer (CAL) [4] system software stack. Figure 1 depicts the memory hierarchy in a CAL system on a heterogeneous CPU-ATI GPU architecture, i.e. local memory, remote memory and application space. The local memory is the high-speed memory on board of the GPU. The remote memory is the set of regions of host memory that are visible to the GPU. Both remote and local memory can be directly accessed by a GPU kernel, but with different latencies. A GPU kernel can directly write data from GPU registers into remote memory using a store instruction, although this operation has a much higher latency than a store on local memory. Remote memory is partitioned into a cached and an uncached portion. For example, in the ATI Stream SDK 2.2, the CAL system reserves 500 MB of cached memory and 1788 MB of uncached memory on an ATI $Radeon^{TM}$ HD5970 device. A good software-pipelining algorithm must place the shared data between CPU and GPU in carefully selected memory regions.

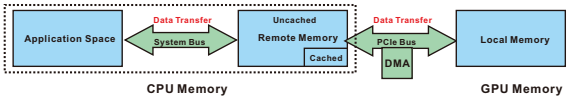

Figure 1: Memory hierarchy in CAL system between CPU and ATI GPU

2.2 Software Pipelining in 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 CPU application space. DGEMM calculates $C = alpha \times A \times B + beta \times C$, where A, B, C are $m \times k, k \times n, m \times n$ matrices respectively. Since in most 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 $A = \{A_1, A_2, \cdots, A_p\}$, $B = \{B_1, B_2, \cdots, B_q\}$, $C = \{C_1, C_2, \cdots, C_{p \times q}\}$, where p and q depend on GPU memory size. For simplicity of presentation, we take p=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: $C_1 = A_1B_1, C_2 = A_1B_2, C_3 = A_2B_1, C_4 = A_2B_2$. 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 [6, 9, 14, 16, 17, 21] (e.g. register blocking). Yang et al. [23] developed a software-pipelining algorithm characterized by four-stage pipelining: *load1* $\rightarrow$ *load2* $\rightarrow$ *mult* $\rightarrow$ *store*. Table 1 explains the related action of each pipelined stage.

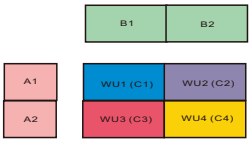

Figure 2: Work units split when p=q=2

Algorithm 1 outlines the four-stage software-pipelining DGEMM algorithm in [23]. The algorithm contains two stages for loading data to GPU local memory (*load1*, *load2*)

Table 1: The four pipelined stages in [23]

load1	copy both A and B from application space to remote memory
load2	copy both A and B from remote memory to local memory
mult	calculate C on GPU device and output them to remote memory
store	copy C from remote memory to application space

and one stage for storing data back to CPU application space (*store*). As data are arranged in different memory regions, the current software-pipelining algorithm loads input matrices A and B into cached remote memory. Thus data reuse can be exploited for executing multiplications. Take the example in Figure 2, if we schedule the execution of work units using the "bounce corner turn" [23], that is in the following order: $WU_1 = \{C_1 = A_1B_1\}, WU_2 = \{C_2 = A_1B_2\}, WU_3 = \{C_4 = A_2B_2\}, WU_4 = \{C_3 = A_2B_1\}$. Every two consecutive work units reuse one of the two input sub-matrices. The resulting sub-matrices of C are allocated to uncached remote memory because a finer blocked matrix multiplication algorithm can exploit register reuse, so that the results are written back for only one time. Another optimization is the usage of double-buffering strategy for overlapping multiplication with the write-back process. The algorithm partitions sub-matrices of C into many smaller blocks and writes two buffers in remote memory in an interleaved way. For each loop j in line 6-9 of Algorithm 1, the *store* dumps one buffer of a block $C_{i,j}$ into application space, while the *mult* calculates the next block $C_{i,j+1}$ and fills the results into the other buffer. Since the *mult* kernel is executed on GPU device in a non-blocking way, it proceeds in parallel with the *store* in every iteration.

Algorithm 1 Four-stage software-pipelining DGEMM algorithm in [23]

Partition: $A = \{A_1, A_2, \cdots, A_p\}, B = \{B_1, B_2, \cdots, B_q\}$, $C = \{C_1, C_2, \cdots, C_{p\times q}\}$
Work units: $WU = \{C_1 = A_1 \times B_1, C_2 = A_1 \times B_2, ...\}$
$C_{i,j}$: *the sub-matrices of* C_j
//
1. bind remote memory for sub-matrices A,B,C
//pre-processing
Allocate workunits using the "bounce corner turn" for exploiting data reuse
//the for-loop is pipelined
2. ***for*** each workunit wu_i **do** $//i = 1, 2, \cdots, p \times q$
//load1
3. copy either A_i or B_i from application space to remote memory
//load2
4. copy either A_i or 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** $//j = 2, 3, \cdots$
//store
7. copy $C_{i,j-1}$ from 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 $C_{i,j}$ from remote memory to application space (also multiplied by beta)
11. ***endfor***

2.3 Motivation

We implement four-stage pipelined algorithm by modifying ACML-GPU library and profiling its execution on the heterogeneous CPU-ATI *Radeon*^{TM} HD5970 GPU architecture. Figure 3 summarizes the time percentage of each stage in four-stage pipelined algorithm. 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 the *mult* kernel occupies the most percentage (about 70%) while the sum of the other three data transfer stages takes up about 30%. The experiments from both [23] and our implementation show that the four-stage software-pipelining algorithm improves performance by about 30%. Intuitively, the overheads of data transfers should be totally hidden by multiplication kernel during the course of the pipelining execution. Our analysis of the pipelining execution reveals an extra data transfer which was not considered in previous software-pipelined designs.

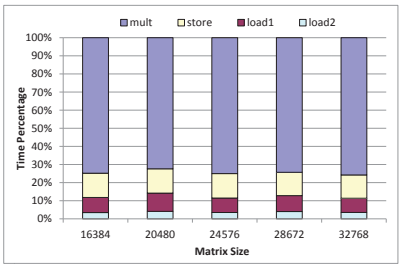

Figure 3: Time percentage of four-stage pipelining algorithm

The resources employed by the algorithm are host memory, GPU and PCIe bus. Operations on these three resources can typically proceed in parallel. Figure 4 outlines the resource usage in each stage of four-stage pipelining. Both *load1* and *store* employ the host memory, and *load2* only needs the PCIe bus to transfer data. The *mult* kernel performs DGEMM on the GPU, and then outputs the results to remote memory through the PCIe bus. The resource usage in each stage suggests straightforward software pipelining that overlaps the data transfers (*load1*, *load2*, *store*) with the *mult* kernel. This pipelining seems beneficial because the *mult* kernel becomes the bottleneck, as Figure 3 shows.

Note that the *mult* kernel in the four-stage pipelining directly stores data from registers to remote memory, causing data transfers through the PCIe bus at the end of each kernel execution. Although these store operations occur only once per kernel invocation, the total amount of data they transferred may be larger than that in *load2*. This happens for example in the DGEMM benchmark included in LINPACK [3] where A, B and C are matrices of size $m \times k, k \times n, m \times n$ respectively, and k is much smaller than both m and n. Therefore, the size of $C(m \times n)$ is larger than the size of A and B $(k \times (m + n))$. Besides, as described in Section 2.2 the latency of the data transfers in *load2* is reduced by exploiting data reuse. We profile the *mult* kernel taken from the four-stage pipelined algorithm and find its floating-point efficiency tops approximately at 50%, which is also observed in ACML-GPU v1.1.2 [2]. As we noted before, Nakasato achieved a maximum of 87% efficiency with a multiplication kernel that stores the results back to local memory in [13]. These results suggest that the difference between local and remote stores is critical to performance, and that the multithreading performed implicitly by GPU hardware and the double buffering approach are not sufficient to hide such long latency. In summary, the efficiency gap between the four-stage pipelined algorithm and the *mult* kernel indicates that there is room for optimizing data movement in the *mult* kernel. Therefore, the software pipelining should be refined to achieve better overlap between computation and data transfers.

3. METHODOLOGY

According to the analysis above, we focus on the *mult* kernel of four-stage software-pipelining algorithm. We address the bottleneck represented by the long latency involved in writing back the result matrix to remote memory. Previous

	Host Memory	GPU	PCIe Bus
load1	X		
load2			X
mult		X	X
store	X		

Figure 4: Resource allocation in each stage of four-stage pipelined algorithm (Algorithm 1), "X" denotes the usage of resources

work [1, 2, 6, 9, 11–14, 16, 17, 19, 21–23] attempted to mitigate this latency via the architectural multi-threading provided by GPU hardware and the double-buffering approach. In this section, we provide instead a solution based on an improved software pipelining. We reduce the write-back latency by employing the image addressing mode available on ATI hardware, and by storing the C matrix to GPU local memory, adding a separate pipelining stage to write back the results to remote memory.

3.1 Addressing Mode

Two addressing modes are available to access GPU local memory: the image addressing mode and the global buffer addressing mode. In global buffer mode, kernels can access arbitrary locations in local memory. In image addressing mode, kernels can only address pre-allocated memory segments. The addresses of image reads are specified as 1D or 2D indices, and reads take place using the fetch units. The programmer must bind these pre-allocated memory segments to the fetch units. For a single kernel invocation, the number of images for reading and writing is 128 and 8, respectively. The maximum dimension of 2D addresses is 8192×8192 elements. Addresses must be contiguous as specified by the 1D or 2D mode and the reads must be invoked with dedicated instructions. Though the image addressing seems a bit complicated to perform, the latency it incurs is much less than the global buffer addressing.

Nakasato [13] described a detailed analysis of choosing blocking factors for DGEMM kernel implementation on Cypress GPU. Both his work and ACML-GPU library indicate that the optimal blocking algorithm calculates 4×4 submatrices in the kernel for satisfying requirements of memory bandwidth and registers. Our multiplication kernel adopts a similar blocking algorithm. Specifically, our kernel invokes 8 input and 8 output samplers using the image addressing mode instead of the global buffer addressing mode.

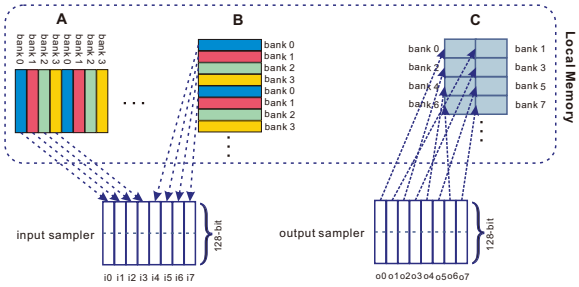

Figure 5: Data partition and sampler mapping of the new multiplication kernel in image addressing mode

As shown in Figure 5, two input matrices in local memory are partitioned into many banks which are loaded for multiplication by every four banks. Each column (row) of A (B) is one bank which is bound to a sampler. For example, every four banks of A are mapped to samplers i0-i3, and every four banks of B are mapped to samplers i4-i7. Since the width of a sampler is 128-bit and a 4×4 sub-block of C matrix is calculated, the output C matrix is partitioned into many two-dimensional banks of 4×2. Algorithm 2 describes the kernel algorithm that calculates a 4×4 sub-block of C matrix in GPU local memory. In order to conserve registers, the algorithm splits banks of both A and B into two parts and loads them in two steps. The resulting sub-blocks of C are stored into local memory, instead of remote memory. Unlike the previous designs, we split the write-back into two stages: the first stores results to local memory, and the second transfers them to remote memory. In the next section, we will show how our new pipelining overlaps the local memory store with the other stages.

Algorithm 2 The algorithm of DGEMM kernel

```
bind banks of both A and B to input samplers
bind banks of C to output samplers
Registers: a[4], b[4], c[8]    //128-bit width
////////////////////////////////////////
c[0 : 8]=0

for (k=0; k<K/4; k++) {
//fetch four banks of two elements from A
load a[0 : 3] ← bank_A[0 : 3][2k]
//fetch two banks of four elements from B
load b[0 : 3] ← bank_B[0 : 1][2k : 2k + 1]
//multiplication
c[0 : 8] + = a[0 : 3] × b[0 : 3]

//fetch four banks of two elements from A
load a[0 : 3] ← bank_A[0 : 3][2k + 1]
//fetch two banks of four elements from B
load b[0 : 3] ← bank_B[2 : 3][2k : 2k + 1]
//multiplication
c[0 : 8] + = a[0 : 3] × b[0 : 3]
} //end for
store c[0 : 8] → bank_C[0 : 8]
```

3.2 Five-stage Pipelining

In our multiplication kernel we output results to GPU local memory instead of remote memory, and a later transfer is required to write the results to remote memory. The difference with respect to previous approaches is the separation of this write-back transfer from the *mult* kernel into a new stage *store1*, which makes us design pipelining of five stages. This separation provides an opportunity to better utilize the DMA engine available in the hardware, and can perform data transfers asynchronously.

Algorithm 3 shows our five-stage pipelining, in which stage *mult* is split into stages *mult1* and *store1*, where *store1* transfers C results from local memory to remote memory. For clarity we rename the old *store* stage as *store2*.

With respect to resources, the *mult1* only occupies the GPU computing cores, whereas the *store1* only occupies the DMA engine. To better overlap with *mult1*, a sub-matrix of C output is divided into smaller ones to transfer. While *mult1* is running, several *store1* operations are executing at the same time. In our implementation, four *store1* stages are in parallel with *mult1* kernel due to the size of sub-matrices calculated. Thus, computation and DMA transfers can proceed in parallel in a fine granularity, similarly to what happens in the double-buffering scheme.

	Host memory	GPU	PCIe Bus
load1	X		
load2			X
mult1		X	
store1			X
store2	X		

Figure 6: Resource allocation in the new five-stage pipelined DGEMM

Figure 6 depicts the new resource allocation. The *mult1* no longer needs PCIe bus, and only uses GPU device. Therefore, *mult1* can now be executed in parallel with *load2* or *store1* without PCIe conflicts. Not only Algorithm 3 pro-

vides a finer-grained pipelining and alleviates the resource conflicts, but it also allows a faster kernel implementation.

3.3 Software Pipelining Design

We propose the use of our five-stage pipelined design not only in order to optimize DGEMM, but as a more general strategy to structure the execution of similar kernels. Our pipelining scheme could serve as the basis to create a reusable software-pipelining programming framework and runtime system, which can be used in a heterogeneous environment based on ATI GPUs. This section outlines how this framework could be implemented.

The framework will provide an API for the users to invoke a five-stage pipelining described in Table 2. Figure 7 illustrates the execution of this pipelining over time, where the five stages are represented with distinct colors. The bars inside the *exec* blocks represent the data transfer stages (*load1*, *load2*, *store1*, *store2*), which are overlapped by the *exec* kernel. As shown in this figure, except for the prologue and epilogue of the pipelining, data transfers are totally overlapped.

The framework will contain a runtime system that allows the users to describe properties of the data transferred (both input and output). These properties characterize the memory access patterns, i.e., whether data are contiguous and whether they are reused. Properties determine the data placement in each pipelining stage:

- For data accessed contiguously, the *exec* kernel should preferably use the image addressing mode, and the runtime system will correspondingly bind those data to the sampling fetch units.
- Those data that have been indicated as subject to reuse should be placed in cached remote memory regions for better reuse.

Algorithm 3 The five-stage software-pipelining DGEMM

Partition: $A = \{A_1, A_2, \cdots, A_p\}, B = \{B_1, B_2, \cdots, B_q\}$, $C = \{C_1, C_2, \cdots, C_{p\times q}\}$
Work units: $WU = \{C_1 = A_1 \times B_1, C_2 = A_1 \times B_2, \cdots\}$
$C_{i,j}$: *the sub-matrices of* C_j
//
1. bind remote memory for sub-matrices A,B,C
//pre-processing
Allocate workunits using the "bounce corner turn"
//the for-loop is pipelined
2. ***for*** each workunit wu_i **do** $//i = 1, 2, \cdots, p \times q$
//load1
3. copy either A_i or B_i from application space to remote memory
//load2
4. copy either A_i or B_i from remote memory to local memory
//mult
5. ***DMAPipeline***$(C_{i,1})$
6. ***for*** each block $C_{i,j}$ **do** $//j = 2, 3, \cdots$
//store2
7. copy $C_{i,j-1}$ from remote memory to application space (also multiplied by beta)
//mult
8. ***DMAPipeline***$(C_{i,j})$
9. ***endfor***
//store2
10. copy the last $C_{i,j}$ from 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** $//k = 2, 3, \cdots$
//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

An interesting experience from our DGEMM kernel optimization is that explicit double-buffering appears to be more efficient for tolerating latency than multi-threading on current ATI GPU architectures, because of the long memory latency to be hidden. Therefore, it is worth implementing double-buffering for reading and writing data in each data transfer stage. In fact, the major source of performance improvement in our optimization is the identification of the inefficiency of the ATI GPU hardware in mitigating the latency due to remote writes. Therefore, we advocate an explicit orchestration of data transfer between local memory and remote memory with a software pipelining approach integrated in a programming framework, although it will increase the complexity of implementing runtime system.

Table 2: The five stages in general software pipelining

load1	copy input data from application space to remote memory
load2	copy input data from remote memory to local memory
exe	perform kernel execution on GPU device and output data to local memory
store1	copy output data from local memory to remote memory
store2	copy output data from remote memory to application space

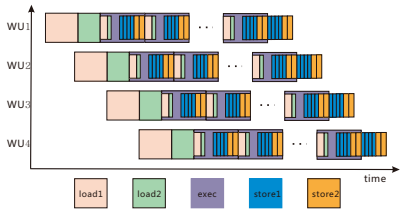

Figure 7: General software pipelining sketch

4. EXPERIMENT RESULTS AND ANALYSIS

4.1 Experimental Setup

Our experiments were performed on the heterogeneous system comprising two Intel Xeon 5650 CPUs and one ATI $Radeon^{TM}$ HD5970 GPU. Table 3 summarizes the configuration parameters of our experimental platform. The CPU provides a peak double-precision performance of 128 GFLOPS. Its memory system is configured with a size of 24GBytes and an aggregated bandwidth of 31 GB/s. The GPU contains two Cypress chips and provides a peak arithmetic throughput of 928 GFLOPS (The calculation is shown below) in double-precision. Its memory size is 2 GBytes and peak memory bandwidth is 256 GB/s. The total double-precision peak performance of the heterogeneous system is 1056 GFLOPS.

GPU performance:

$$928GFLOPS = 725MHz(frequency) \times 2(\#DoublePrecisionFPRates) \times 320(\#StreamCores) \times 2(\#chips)$$

Table 3: Configuration of the experimental platform

Processors	Xeon X5650	$Radeon^{TM}$ HD5970
Model	Westmere-EP	Cypress
Frequency	2.66GHz	725MHz
#chips	2	2
DP	128 GFLOPS	928 GFLOPS
DRAM type	DDR3 1.3GHz	GDDR5 1.0GHz
DRAM size	24GB	2GB
DRAM bandwidth	31.2 GB/s	256 GB/s
PCIe2.0	x16, 8 GB/s	
Programming	icc + openmpi	ATI Stream SDK 2.2

We evaluate multiple implementations of DGEMM on our experimental platform, and the abbreviations are described below in order of incremental optimizations:

- ACML-GPU: It is the ATI DGEMM library, which uses direct pipelining strategy among work units to

hide the overheads caused by writing result C matrix from remote memory to application space. The *mult* kernel uses global buffer addressing mode.

- 4-stage pipelining: This is a DGEMM implementation of Algorithm 1. It improves ACML-GPU by overlapping the load operations of input matrices and using double-buffering strategy within a work unit better overlapping write-back of C matrix. It is realized according to [23].
- 5-stage pipelining: This is our new implementation that focuses on orchestrating the data placement through the memory hierarchy. It also selects a better addressing mode and data placement for the output C matrix.
- HDGEMM: While the above three implementations only exploit the arithmetic throughput of the GPU, this one implements a hybrid DGEMM, where CPU and GPU perform the workload cooperatively. This is also our best DGEMM implementation. It partitions matrices between the CPU and the GPU, and adaptively balances the workload between them using a strategy proposed in [23]. Our experiments use two processes, each comprising one computing element (CE) (one CPU and one GPU chip).

Table 4: DGEMM matrix dimensions and the corresponding storage sizes in Gigabytes used in the following experiments, and the number of work units

GB / k	m=n				
	16384	**20480**	**24576**	**28672**	**32768**
1536	2.55	3.86	5.44	7.28	9.40
2048	2.68	4.03	5.64	7.52	9.66
4096	3.22	4.70	6.44	8.46	10.74
#workunits	6	12	12	20	24

Table 4 shows the matrices used in our experiments, with the order (m,n,k) and corresponding size in Gigabytes. We keep m equal to n 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. Besides, the number of work units used in our experiments is given in the last row. Since we divide matrices in m and n dimension, the number of work units is independent of k size. 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 size in x axis represents different m (n) values.

4.2 Results

We first report the overall performance of the four DGEMM implementations on our experimental system. Figure 8 plots the performance and efficiency with different matrix sizes, as given in Table 4. The performance of DGEMM is measured in terms of GFLOPS i.e. the number of floating-point operations executed per second (in billions). Efficiency is defined as the ratio between the performance achieved and the machine's theoretical peak performance.

Each group of five bars represents the corresponding performance for each k value (k=1536, 2048, 4096 in the x axis) with five m (n) values, as shown in Table 4. The numbers on the left y axis are the floating-point performance in GFLOPS. From bottom up the segments of each bar indicate the performance increment. For each bar, the bottom segment represents the baseline program ACML-GPU, the second one above represents performance improvement of the four-stage pipelining implementation over ACML-GPU, the third one shows performance improvement of five-stage pipelining over four-stage pipelining, the performance gains of hybrid CPU-GPU HDGEMM are depicted on the top. For simplicity, Figure 8 plots two efficiency lines (values shown in the right y axis), which correspond to the best GPU-only (5-stage-pipelining) DGEMM and HDGEMM respectively.

Our HDGEMM achieves a maximum performance of 844 GFLOPS and an efficiency of 80% (when <m,n,k> = <16384, 16384, 4096>) on the whole system with two Cypress GPU chips and two six-core CPUs. The 5-stage-pipelining DGEMM reaches a maximum performance of 758 GFLOPS and an efficiency of 82% (when <m,n,k> = <16384, 16384, 4096>) on two Cypress GPU chips. In a comparison against ACML-GPU, our five-stage pipelined implementation improves performance by a factor of 2 on average. HDGEMM further improves performance by 10%-20% by utilizing the computing power of multi-core CPUs.

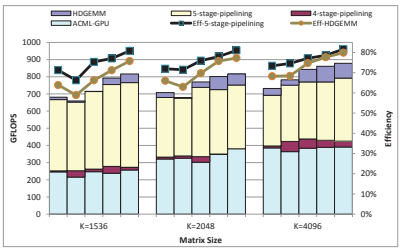

Figure 8: Our optimized DGEMM performance and efficiency on two GPUs and/or two CPUs

When k is fixed, all four implementations show an increase in both performance and efficiency with larger matrices. There are few cases with abnormal behavior (e.g. m=n=10240). It is because the matrix size is not a multiple of the optimal block size for data transfer and kernel execution. As we expected, comparing the average performance among different k sizes, the performance improvement of our five-stage pipelined implementation over ACML-GPU drops as the matrix scale becomes larger. When k is 1536, 2048 and 4096, the speedups are 2.9X, 2.1X, and 1.9X respectively. This is because the ratio of data transfer to kernel execution decreases with larger problem size. Since our software pipelined optimization focuses on data transfers, the performance improvements will be more obvious when data transfers take large portion of the implementation. On larger datasets, the computation absorbs a larger fraction of the DGEMM execution time, and the effect of data-transfer optimizations is less apparent. As Figure 8 shows, HDGEMM exhibits higher performance improvements with larger matrix sizes. That happens because the CPU delivers higher performance on large matrices, increasing the overall performance of HDGEMM.

In order to highlight the effect of the proposed optimizations in this work, we isolate sources of noise (e.g. bandwidth contention, which will be reported in the next section) that are present when multiple GPU chips are employed in the heterogeneous system. We perform experiments on one GPU chip and do not assign any computing load to CPU, which is only in charge of data transfers. Assuming k=2048, Figure 9 compares the performance of the three GPU-only DGEMM implementations, i.e., ACML-GPU, 4-stage pipelining, 5-stage pipelining. Compared with ACML-GPU, the 4-stage pipelining improves performance

by about 30% through pipelining *store2* within a work unit and reusing data.

Our 5-stage pipelining implementation improves performance by 74% employing a better data placement, the image addressing mode, and a finer-grained pipeline. It explicitly leverages the DMA engine to pipeline the result write-back stage. Finally it achieves 408 GFLOPS with an efficiency of 88% on one Cypress GPU chip. This implementation only shows a 5% performance loss compared with an execution of the kernel without CPU-GPU transfers (94% efficiency). It demonstrates that our pipelining optimizations achieve almost complete success in mitigating data transfer latencies.

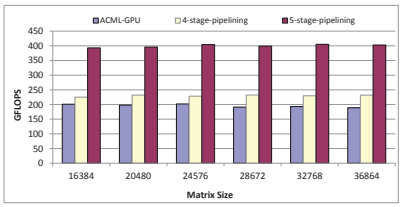

Figure 9: Performance improvements on one GPU chip

In Section 2, Figure 3 shows that the three data transfer stages (*load1*, *load2*, *store*) are responsible for about 30% of the total execution time (statistics do not include the implicit data transfer *store1* taking place in the *mult* stage). We refine the pipelining by separating *store1* from the *mult*. Our approach achieves a better pipelining. In fact, after counting the added *store1* stage, data transfers account for more than 40% of the total execution time. The new execution time fractions for each data transfer stage (*load1*, *load2*, *store1*, *store2*) are in Figure 10. Only the data transfer stages are given, and y axis represents the ratios between the time of each stage and that of the total data transfer. *Store1*, which takes about 55% of the total data transfer time, is the major part of the four data transfers.

From Figure 9, the performance improvement of the five-stage pipelining over the four-stage one is 74% on average; this number is larger than all the data transfer percentage of DGEMM (about 43%). This is because our optimizations improves not only data transfers, but also the performance of the multiplication kernel. Besides, Figure 9 shows that our optimized DGEMM performance is quite stable through all the matrix sizes. This property lays a good foundation for good scalability of our DGEMM implementation on multiple chips of both CPUs and GPUs. However, the stable trend on one GPU chip is different from that shown in Figure 8, where the HDGEMM efficiency line occurs below the 5-stage-pipelining line. 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 implementation on the heterogeneous system also achieves the maximal efficiency of 80%, counting data transfer between CPU and GPU. In this section we will investigate: (i) How much optimization room is left beyond our pipelining optimizations on such a heterogeneous architecture. (ii) What about the intra-node scalability of HDGEMM when scaling to multiple CPUs and GPUs. Note that we focus on multiple GPU chips on the same board that share the same CPU memory, because it is one trend that a heterogeneous system integrates multiple GPU chips or cards into one board (node). Our analysis attempts to figure out some architectural constraints on the performance of our heterogeneous algorithm. One more point to be clarified is that we are not going to evaluate very large scale matrices distributed on many nodes. On one hand, a call to DGEMM routine usually happens after the higher level algorithm has partitioned the original problem size across nodes. Thus, each node executes DGEMM independently. On the other hand, our software pipelining optimization is customized to a tightly coupled a CPU-GPU heterogeneous architecture. Therefore, an evaluation on a large cluster system is out of the scope of this paper.

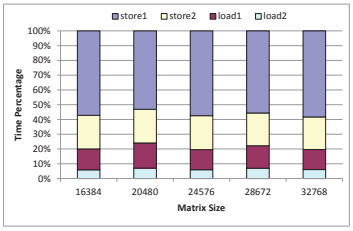

Figure 10: Time percentage of the four data transfer stages

4.3.1 Performance Gap

Among the five stages of the pipelining, the *mult1* stage determines the highest performance that DGEMM can achieve. Nakasato [13] optimized the DGEMM performance and reported the highest efficiency of 87% on an ATI HD5870 GPU that uses only one Cypress chip. Through the use of the image addressing mode (instead of the global buffer addressing mode) for the result matrix write-back, we achieve a higher efficiency of 94% on one Cypress chip.

Figure 11 compares the efficiency of the *mult1* kernel alone, the 5-stage pipelining implementation on one GPU chip, the 5-stage pipelining implementation on two GPU chips, and the HDGEMM implementation on a full system of two CPUs and two GPU chips. For each set of experiments, we show the average efficiency. DGEMM invokes the *mult1* kernel for multiple times in every execution, and the *mult1* kernel performance plotted is the average value over these times. 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 floating-point efficiency of the CPU DGEMM library. The difference between the kernel and 5-stage pipelining is whether the data transfer between CPU and GPU is counted. Results show that, when running on one chip, the performance of the 5-stage pipelining implementation on one GPU is 6% lower than the kernel running alone, due to the data transfers. There are two reasons for the performance drop. First, the prologue and epilogue of the pipelining cannot be hidden, and they absorb around 3% of the total execution time (measured in our experiments). Second, as shown in Figure 6, there are still resource conflicts within the pipelining. During its execution, there is host memory contention between *load1* and *store2*, and PCIe bus contention between *load2* and *store1*.

When scaling within a node, contention of shared resources will limit performance. Figure 11 shows the intra-node scalability of HDGEMM with more GPUs and/or CPUs as well. Running 5-stage pipelining DGEMM on two GPU chips (5-

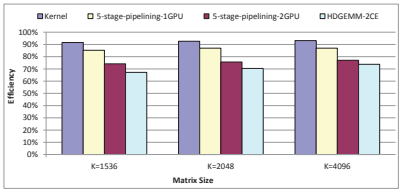

Figure 11: The scalability of our optimized DGEMM when scaling to multiple CPUs and/or GPUs

stage-pipelining-2GPU), the efficiency drops by 11% with respect to 5-stage-pipelining-1GPU. Moreover, when we extend DGEMM to a system with two computing elements (HDGEMM-2CE), the efficiency further drops by 5% from 5-stage-pipelining-2GPU. We believe that resource contention is the main reason influencing DGEMM scalability. In the following sections, we will focus on HDGEMM intra-node scalability, and discuss host memory contention and PCIe contention in more details.

4.3.2 Contention on Multiple GPUs

Many state-of-the-art heterogeneous systems include accelerators (i.e., GPUs, ClearSpeed cards, Tilera) that are attached to the CPU through a PCIe bus that resides on the motherboard. Frequently there are multiple PCIe slots, each supporting one GPU board; moreover, some GPU boards host multiple GPU chips, e.g., the ATI $Radeon^{TM}$ HD5970 and the NVIDIA Tesla S1070. It is therefore valuable to analyze the scalability of our design on a system comprising multiple GPU chips.

As motherboards are concerned, the lane allocation among PCIe slots is different among models. Some motherboards support x16 + x16 combination, while others only support x8 + x8. If the combination is x16 + x16, there is no PCIe contention between different GPU boards, thus the intra-node scalability will not be influenced by PCIe contention. However, if a lane is divided between two GPU boards as in an x8 + x8 combination, which is more familiar, the PCIe usage is the same as two chips within one GPU board with an x16 slot.

In this paper, we focus on experimental conditions where PCIe contention exists, i.e., multiple GPU boards affected by the limited lane number, or multiple GPU chips within one GPU board. It makes sense to assume that the scalability on a system with multiple GPU chips on a board is similar to the one with multiple single-chip GPUs.

Due to the limitations of our experimental platform, we run two processes, each of which is in charge of one CPU and one GPU chip. The experiments profile the changes in available bandwidth from one GPU chip to two GPU chips, and analyze bandwidth contention between the two GPU chips to predict its scalability with more GPUs. For ease of comparison, each process of 5-stage-pipelining-2GPU runs with the same problem scale as that of 5-stage-pipelining-1GPU. From Figure 6, the bandwidth contention exists on both PCIe bus and host memory.

In order to focus on the bandwidth changes, the measured bandwidth of 5-stage-pipelining-2GPU is normalized to that of 5-stage-pipelining-1GPU. First, we consider PCIe contention during the execution of *load2* and *store1*. The reduction of average bandwidth is shown in Figure 12 with the normalized values in the y axis. As shown in this figure, *load2* and *store1* only achieve 89% and 56% of the PCIe bandwidth in 5-stage-pipelining-1GPU, respectively. We see

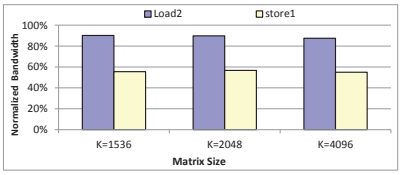

Figure 12: PCIe bandwidth of 5-stage-pipelining-2GPU normalized to that of 5-stage-pipelining-1GPU

significant a decrease in *store1*, 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, while *mult1* is running, several *store1* operations are executing at the same time (4 *store1* stages in our implementation). During *mult1* execution, the majority of PCIe bandwidth is consumed by *store1*, which reaches high occupancy even on one GPU chip. Therefore, when scaling to two GPU chips, PCIe bus contention becomes more severe. However, the bandwidth of PCIe bus available in the direction from the CPU to the GPU (*load2*) does not suffer as much as *store1*. That is because *load2* is pipelined with *mult1* among work units, so that the requests on the PCIe bus are not as frequent as *store1*. Besides, the size of matrices transferred by *load2* is $(m + n) \times k$, while the size of matrix transferred by *store1* is $m \times n$. Since k is much less than n in our experiment, the former puts less pressure on the PCIe bus.

In addition to PCIe bus contention, there is also contention on the host memory, since data is copied between the two regions of host memory (application space and remote memory). Figure 13 shows that the relative host memory bandwidth of *load1* and *store2* normalized to that of 5-stage-pipelining-1GPU. 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, while the bandwidth reduction is smaller, because the frequency of *store2* is much less than that of *store1*.

Profiling shows that stages *load1*, *load2*, *store1* and *store2* are almost overlapped completely with *mult1* kernel in 5-stage-pipelining-1GPU. However, the bandwidth contention on the two GPU chips leads to an 11% efficiency loss: see 5-stage-pipelining-2GPU in Figure 11. Since two processes contend the shared resources (PCIe bus and host memory), some data transfers are prevented from overlapping with the *mult1* kernel. This reduces the performance. As the number of CPU or GPU chips increases, bus requests will become more frequent, decreasing efficiency even more because of resource contention. Our experimental results motivate the following two observations:

- Observation 1: **Due to the contention of PCIe bus, DGEMM hits its limitation on multiple GPUs with restricted number of lanes.** In DGEMM implementation, both *load2* and *store1* compete for PCIe bandwidth. As shown in Figure 10, these two phases occupy more than 60% of the total data transfer time. Our experimental results in Figure 12 show a significant decrease of bandwidth and 11% performance drop only scaling to two GPU chips. The situation would be worse if more GPUs share PCIe bandwidth.
- Observation 2: **DGEMM on multiple GPUs will not benefit much by improving host memory bandwidth.** Although both *load1* and *store2* consume host memory, it seems they are not quite sensitive to the contention from Figure 13. Besides, their

execution time is not the major part of the total data transfer time (Figure 10). For a small part of applications, the usage of pinned memory would help avoid both *load1* and *store2*. However, it is a must that no data rearrangement is required, and data should fit in the limited pinned memory space. Our work proves that data transfer overheads can also be mitigated by algorithmic optimizations with less limitation.

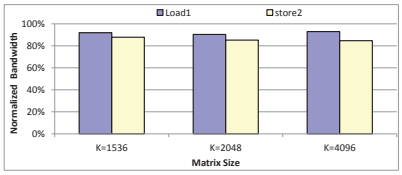

Figure 13: Host memory bandwidth of 5-stage-pipelining-2GPU normalized to that of 5-stage-pipelining-1GPU

4.3.3 Contention on Hybrid CPUs and GPUs

In our platform, the Intel Xeon CPU provides a peak arithmetic throughput of 128 GFLOPS, which contributes 12% of the whole system performance. CPUs should not be neglected when optimizing compute-intensive algorithms like DGEMM. In HDGEMM, matrices are first equally split into two parts, each of which is calculated by a distinct computing element (one CPU and one GPU). Within each computing element, we adopt the algorithm in [23] to partition the workload between the CPU and the GPU. HDGEMM-2CE improves the performance by 6% on average over 5-stage-pipelining-2GPU (see Figure 8), but its efficiency decreases by 5% (see Figure 11). In this section we explain why.

We profile the performance contributed only by CPU (denoted as CPU-HDGEMM) in Figure 14 when HDGEMM-2CE is executed. For comparison, we run a CPU-only DGEMM implementation (denoted as PureCPU), which calculates the same matrix size as CPU-HDGEMM in HDGEMM-2CE. From this figure, CPU-HDGEMM shows a performance loss of 22% compared to PureCPU. The comparison illustrates that HDGEMM prevents CPU from achieving its peak computing performance. We believe that CPU-HDGEMM performance is influenced by the GPU operations *load1* and *store2*. Data transfers to/from GPU share the same application space with CPU's DGEMM calculation. For the same reasons as discussed with 5-stage-pipelining-2GPU, host memory contention does not affect DGEMM performance in the GPU part of HDGEMM-2CE. However, the sharing of application space seriously influences DGEMM performance when executing on CPUs. We further observe that:

- Observation 3: **Host memory bandwidth is an important factor to HDGEMM performance.** As CPU arithmetic throughput increases, host memory contention will have greater impact on the overall performance of HDGEMM. Some applications might alleviate this contention by employing pinned memory.

Another minor reason for the efficiency degradation is the imbalanced workload partition between CPU and GPU. In our adopted partition strategy, a heuristic algorithm in [23] is used to search an appropriate split ratio between CPU and GPU, making the execution time difference less than a threshold. We take 0.1 seconds as the threshold in this paper, which is an empirically optimal value selected by multiple iterations of experiments. Figure 15 plots the execution

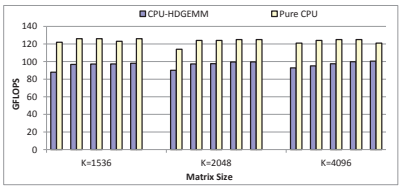

Figure 14: Comparison of performance that CPU contributes in HDGEMM and CPU-only DGEMM

time difference between CPU and GPU in different matrix sizes. We take CPU execution time as reference, and the time difference is calculated by $(time_{gpu} - time_{cpu})/time_{cpu}$. 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%).

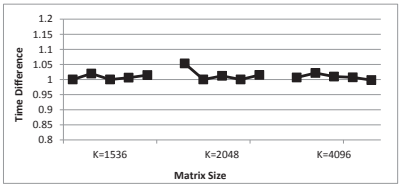

Figure 15: Relative time difference between CPU and GPU

5. RELATED WORK

The most related work includes both Nakasato's kernel optimization [13] and Yang's software-pipelining optimization [23]. Table 5 summarizes the optimization strategies used in the three DGEMM programs. All the optimizations improve performance over the baseline ACML-GPU library [2]. Until now, our DGEMM achieves the highest performance on the heterogeneous CPU and ATI GPU architecture. In addition, we disclose some experimental observations on the shared resources (PCIe bus and host memory) contention on a heterogeneous system.

Table 5: Comparison with [13] and [23]

	Nakasato [13]	Yang [23]	Ours
optimizations			
image addressing for C	no	no	yes
local memory for C	yes	no	yes
pipelining	no	four-stage	five-stage
data reuse	no	yes	yes
double-buffer in local memory	no	no	yes
double-buffer in remote memory	no	yes	yes
performance (maximal GFLOPS (floating-point efficiency))			
kernel	470 (87%)	248 (53%)	436 (94%)
one GPU chip	~300 (55%)	234 (50%)	408 (88%)
two GPU chips	-	438 (47%)	758 (82%)

Some other work optimized DGEMM assuming that the matrices have already been resident in GPU on-board memory. AMD's Accelerated Parallel Processing Math Libraries (APPML) v1.4 [1] in OpenCL language provides GPU-only DGEMM kernel, according to the test our kernel is more efficient than it. GATLAS auto-tuner in [11] makes use of auto-tuning method to increase the portability among different GPU architectures and is meant to be used in realistic applications. However, it still solves DGEMM which matrices are resident in GPU on-board memory. Thus, there is no direct way to call GATLAS in realistic applications so far with large datasets. V. Volkov and J. Demmel implemented one-sided matrix factorizations (LU, QR, etc.) on a hybrid CPU-GPU system in [21], they divided the factorization processes to CPU and GPU separately. Matrix-matrix multiplication in that case still uses data stored in

GPU memory without data transfer. MAGMA [14] develop a dense linear algebra library similar to LAPACK [5] for heterogeneous architecture, it has been implemented only for NVIDIA GPU so far. Therefore, our optimizations will provide a possible solution for MAGMA when extending to ATI GPU. Other math library implementations for heterogeneous architecture include hybrid Jacobi [20], model-based heterogeneous FFT [15], etc. Most of the work did not carefully optimize data transfer, but parallelized the computation on CPU with GPU computation instead. The software-pipelining approach may be a complement to improve the performance of hybrid libraries.

6. CONCLUSION

We analyze the state-of-the-art implementation of the DGEMM algorithm when running on a heterogeneous system comprising a CPU and an ATI GPU, and find sources of inefficiency. We propose a more optimized five-stage software-pipelined design, and provide an implementation that exploits the image addressing modes available on the ATI hardware. Our design mitigates better the latencies of CPU-GPU data transfers, delivering 408 GFLOPS (with 88% floating-point efficiency) on one Cypress GPU chip, 758 GFLOPS (82% efficiency) on two Cypress GPUs (i.e., the entire GPU board), and 844 GFLOPS (80% efficiency) on a system comprising two Intel Westmere-EP CPUs and an ATI $Radeon^{TM}$ HD5970 GPU. We show that the use of multiple GPUs on the same node increases the performance but achieves lower efficiency, due to the contention of shared resources, in particular the PCIe bus and the host memory. We believe that hardware designers should focus on reducing the cost of such contention instances if they desire their hardware to achieve higher degrees of efficiency with DGEMM-like workloads.

7. ACKNOWLEDGMENTS

We would like to express our gratitude to Dr. Daniele Paolo Scarpazza at D.E. Shaw Research for helping us revise this paper, and Dr. Udeepta Bordoloi for the comments. This work is supported by National 863 Program (2009AA01A129), the National Natural Science Foundation of China (60803030, 61033009, 60921002, 60925009, 61003062) and 973 Program (2011CB302500 and 2011CB302502).

References

[1] *AMD Accelerated Parallel Processing Math Libraries.*

[2] *AMD Core Math Library for Graphic Processors.*

[3] HPL - a portable implementation of the high-performance linpack benchmark for distributed-memory computers.

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

[5] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. LAPACK: A portable linear algebra library for high-performance computers. In *Supercomputing '90. Proceedings of*, pages 2 –11, nov 1990.

[6] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293 –312, feb. 2005.

[7] J. Dongarra, P. Beckman, T. Moore, et.al. The international exascale software project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, Feb. 2011.

[8] J. J. Dongarra, J. Du Croz, S. Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, Mar. 1990.

[9] K. Goto and R. Geijn. Anatomy of high-performance matrix multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.

[10] T. B. Jablin, P. Prabhu, J. A. Jablin, N. P. Johnson, S. R. Beard, and D. I. August. Automatic CPU-GPU communication management and optimization. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, pages 142–151, New York, NY, USA, 2011. ACM.

[11] C. Jang. GATLAS: GPU automatically tuned linear algebra software.

[12] Y. Li, J. Dongarra, and S. Tomov. A note on auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[13] N. Nakasato. A fast GEMM implementation on the Cypress GPU. *SIGMETRICS Perform. Eval. Rev.*, 38(4):50–55, Mar. 2011.

[14] R. Nath, S. Tomov, and J. Dongarra. An improved MAGMA GEMM for Fermi graphics processing units. *Int. J. High Perform. Comput. Appl.*, 24(4):511–515, Nov. 2010.

[15] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka. An efficient, model-based CPU-GPU heterogeneous FFT library. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1 –10, april 2008.

[16] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 73–82, New York, NY, USA, 2008. ACM.

[17] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-m. W. Hwu. Program optimization space pruning for a multithreaded GPU. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[18] M. Silberstein, A. Schuster, and J. D. Owens. Accelerating sum-product computations on hybrid CPU-GPU architectures. In W. W. Hwu, editor, *GPU Computing Gems*, volume 2, chapter 36, pages 501–517. Morgan Kaufmann, Oct. 2011.

[19] G. Tan, Z. Guo, M. Chen, and D. Meng. Single-particle 3d reconstruction from cryo-electron microscopy images on GPU. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 380–389, New York, NY, USA, 2009. ACM.

[20] S. Venkatasubramanian, R. W. Vuduc, and n. none. Tuned and wildly asynchronous stencil kernels for hybrid CPU/GPU systems. In *Proceedings of the 23rd international conference on Supercomputing*, ICS '09, pages 244–255, New York, NY, USA, 2009. ACM.

[21] V. Volkov and J. W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[22] H. Wong, M.-M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *Performance Analysis of Systems Software (ISPASS), 2010 IEEE International Symposium on*, pages 235 –246, march 2010.

[23] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu. Adaptive optimization for petascale heterogeneous CPU/GPU computing. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 19 –28, sept. 2010.