Performance Study of DGEMM on CPU/ATI GPU Hybrid Architectures (Postprint)

Authors: Jiajia Li, Li Xingjian, Tan Guangming

Date: 2017-03-10T00:00:00+00:00

Abstract

This paper reports our work on optimizing double-precision matrix multiplication (DGEMM) on CPU/ATI GPU hybrid architectures. In real-world applications, data transfer between the CPU and graphics processor (GPU) is a key factor affecting performance. As software pipelining can reduce data transfer overhead, we propose three software pipelining algorithms: Double Buffering, Data Reuse, and Data Placement. On AMD' s ATI HD5970 graphics processor, the optimized DGEMM achieves a performance of 758 GFLOP/s with a corresponding efficiency of 82%, which is double the performance of ACML-GPU v1.1. On a heterogeneous system consisting of Intel Westmere EP and ATI HD5970, the performance reaches 844 GFLOP/s with an efficiency of 80%. We further examine the scalability of DGEMM across multiple CPUs and multiple GPUs, and conduct a detailed analysis of architectural impact factors. The analysis indicates that contention on the PCIe bus and memory bus is a significant factor contributing to performance degradation of programs on heterogeneous systems.

Full Text

Performance Study of DGEMM on CPU/ATI GPU Hybrid Architectures

Jiajia Li, Xingjian Li, Guangming Tan

Abstract

This paper reports our work on optimizing double-precision matrix multiplication (DGEMM) on CPU/ATI GPU hybrid architectures. In real applications, data transfer between CPU and graphics processor (GPU) is a critical performance factor. Since software pipelining can reduce data transfer overhead, we propose three software pipelining algorithms: Double Buffering, Data Reuse,

and Data Placement. On AMD' s ATI HD5970 GPU, the optimized DGEMM achieves 758 GFLOP/s, corresponding to 82% efficiency, which is twice the performance of ACML-GPU v1.1. On a heterogeneous system composed of Intel Westmere EP and ATI HD5970, performance reaches 844 GFLOP/s with 80% efficiency. We further investigate the scalability of DGEMM across multiple CPUs and GPUs, analyzing architectural influences in detail. Our analysis reveals that contention on PCIe bus and memory bus is a significant factor degrading program performance on heterogeneous systems.

Keywords: High-performance computing, GPU CAL, Matrix multiplication

1 Introduction

Double-precision matrix multiplication is a crucial factor affecting the performance of numerous scientific and engineering applications. Key numerical algorithms such as BLAS Level 3 [1] and LU decomposition [2] rely on high-performance DGEMM implementations. The HPL benchmark [3], used for ranking the world' s supercomputers, consists of dense matrix LU decomposition. Since DGEMM performance heavily depends on hardware, many processor vendors have developed machine-specific implementations, such as Intel' s MKL and AMD' s ACML, continuously optimizing BLAS libraries for their multi-core processors. Graphics processors (GPUs) offer an order of magnitude higher floating-point peak performance than general-purpose CPUs—for instance, AMD HD5970 delivers 928 GFLOP/s double-precision performance, while NVIDIA Tesla C2070 provides 515 GFLOP/s peak double-precision performance. DGEMM is compute-intensive with regular memory access patterns, making it well-suited for GPU optimization. Indeed, numerous works have optimized DGEMM on GPUs [11-17,19-21,23-27]. Most previous research assumes matrices already reside in GPU memory, focusing on optimizing DGEMM implementation on GPUs. In current systems, GPUs connect to CPUs via PCIe bus as accelerator components. In practical applications, DGEMM input matrices initially reside in CPU main memory. Since GPUs can only access data stored in GPU memory, data transfer between CPU and GPU is required before computation. Due to limited GPU memory capacity, large-scale data requires multiple transfers between CPU and GPU.

In CPU/GPU heterogeneous systems, data transfer between memory hierarchy levels significantly impacts DGEMM performance. From a system perspective, the memory hierarchy has two levels: CPU main memory and GPU memory. First, CPU transfers data to GPU memory via PCIe bus, then GPU shaders fetch data from memory for computation. GPU HD5970 has a peak memory bandwidth of 256 GB/s, while PCIe bus peak bandwidth is only 8 GB/s. Although DGEMM is compute-intensive, this bandwidth disparity remains a bottleneck for overall DGEMM performance on heterogeneous systems. N. Nakasato noted in [15] that if CPU-ATI Cypress GPU data transfer time is included in DGEMM total time, efficiency drops from 85% to 55%. AMD' s GEMM optimization library for CPU-ATI GPU heterogeneous systems exhibits

similar efficiency degradation.

Optimizing DGEMM and analyzing future hybrid architecture trends require quantitative analysis of each memory hierarchy level: understanding how much impact current CPU-GPU heterogeneous architectures and memory hierarchies have on overall DGEMM performance. While previous works have optimized DGEMM on GPUs, few have conducted quantitative analysis, particularly for CPU/GPU heterogeneous systems.

This paper addresses these issues through a study of multi-core CPU/Cypress GPU heterogeneous systems. By designing new algorithms to reduce data transfer overhead between memory hierarchy levels, we achieve a fast DGEMM implementation on this heterogeneous system.

Our contributions are threefold: * We quantitatively analyze DGEMM performance on heterogeneous architectures, with detailed analysis of CPU-GPU data transfer processes. Our analysis shows that data transfer accounts for 40% of total DGEMM time, rather than the 20% reported in previous work, demonstrating that data transfer overhead significantly impacts DGEMM performance. * We propose novel pipelining algorithms for large-scale DGEMM. The three optimization methods are: Double Buffering, Data Reuse, and Data Placement. The optimized DGEMM achieves 408 GFLOP/s (88% efficiency) on a single Cypress GPU, more than twice the performance of ACML-GPU v1.1. On ATI HD5970, optimized DGEMM reaches 758 GFLOP/s (82% efficiency). The hybrid DGEMM on Intel Westmere EP/ATI HD5970 achieves 844 GFLOP/s, representing 80% of peak performance. * We analyze resource contention when scaling DGEMM to multiple CPUs and GPUs, focusing on PCIe contention and memory contention. Our analysis reveals that resource contention (PCIe and memory) is a primary bottleneck for scalability, though software approaches alone cannot completely eliminate its impact on overall DGEMM performance.

2 Background

Our performance optimization targets the ATI Evergreen GPU architecture; Cypress is the flagship GPU in the Evergreen family. The ATI HD5870 card contains one Cypress chip, while HD5970 integrates two Cypress chips. This section introduces key Cypress features—memory hierarchy at the ATI CAL [6] software level. We use DGEMM from the ACML-GPU library as our baseline and foundation for performance optimization, conducting detailed analysis to identify optimization opportunities.

2.1 Cypress GPU

An Evergreen GPU chip integrates multiple compute units, a control unit (also called thread dispatch unit), memory controllers, and DMA engines. To maximize floating-point operation throughput, the Cypress GPU microarchitecture employs Single Instruction Multiple Data (SIMD) and Very Long Instruction

Word (VLIW). Since optimizing DGEMM kernels on GPU is not the focus of this paper, we do not describe GPU microarchitecture details here. We use the kernel from [15] with 80% efficiency. Briefly, a Cypress card at 725MHz achieves 2.32 TFLOP/s single-precision peak performance; double-precision performance is 1/5 of single-precision, yielding 464 GFLOP/s double-precision peak performance at 725MHz.

[Figure 1: see original paper] illustrates the memory hierarchy of the CPU/ATI GPU heterogeneous architecture at the CAL software system level. Below we detail memory hierarchy features relevant to program optimization in CAL systems: * CAL divides physical memory into local and remote storage. Local storage refers to GPU memory—high-speed memory on the graphics card. Remote storage refers to physical memory space not on the graphics card but accessible by GPU (i.e., part of CPU main memory). Both local and remote storage can be read by GPU kernels. In other words, GPU kernels can write data from GPU registers back to either GPU memory or remote storage. Operations on remote storage have higher latency than local storage, resulting in lower performance. Remote storage is further divided into cached and uncached portions. For example, CAL partitions HD5970 remote storage into 1788MB uncached and 500MB cached storage. Therefore, the choice of storage space for shared data between CPU and GPU affects program performance. * In CAL applications, initial data resides in CPU application space. GPU kernels require two data transfers before using data: between application space and remote storage, and between remote storage and local storage. One optimization method uses system pinned memory—applications directly transfer data from CPU application space to remote storage space for DMA transfer. This eliminates data transfer from application space to remote storage. Some applications using CUDA [5] effectively improve performance through this method. CAL also supports pinned memory technology, but with limitations on size and other aspects. Therefore, proper organization between memory hierarchy levels (such as pipelining algorithms) is essential to reduce data transfer time on PCIe bus.

2.2 DGEMM

This section describes the large-scale DGEMM algorithm on CPU/ATI GPU heterogeneous systems, where original data is stored in CPU application space. DGEMM computes C = alpha × A × B + beta × C, where A, B, and C are matrices of dimensions m×k, k×n, and m×n respectively. In practical DGEMM applications, these matrices are too large to fit entirely in GPU memory. Therefore, matrices are partitioned into multiple sub-matrix blocks, with several blocks transferred to GPU memory at a time for partial DGEMM computation. The partitioning algorithm divides A, B, and C into {A , A , ···}, {B , B , ···}, and {C , C , ···} respectively, where p and q depend on GPU memory size. Using p=2, q=2 as an example, [Figure 2: see original paper] shows this partitioning creates four independent C sub-matrices: C =A ×B , C =A ×B ,

C =A ×B , C =A ×B . These four matrix block operations are independent and can be computed in parallel. For each sub-matrix multiplication, we transfer the required A and B sub-matrices to GPU memory, where the DGEMM kernel further divides these sub-matrices into smaller blocks [9-13] (e.g., cache blocking and register blocking).

Based on the ATI CAL system memory hierarchy, large-scale DGEMM implementation is shown in Algorithm 1. Remote memory serves as shared space between CPU and GPU; data transfer between CPU application space and GPU memory (local storage) must pass through remote storage (line 1). In Algorithm 1 pseudocode, loading data to GPU involves two steps (load1 and load2), while writing data back to CPU memory is one step (store). In fact, line 5' s DGEMM kernel operation implicitly includes another data write-back operation—sub-matrix C data is written from GPU registers back to remote storage.

Algorithm 1: Original DGEMM Implementation

```
Partition: A={A , A , ⋯, A }, B={B , B , ⋯, B_q}, C={C , C , ⋯, C_{p×q}}
Work unit: WU={C =A ×B , C =A ×B , ...}

1. bind remote memory for sub-matrices A, B, C
2. for each workunit wu_i do // i=1,2,⋯,p×q
   // 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 from remote memory to application space (also multiply by beta)
7. endfor
```

We use Algorithm 1 as our baseline for performance comparison. Below we present detailed profiling of DGEMM runtime components on heterogeneous CPU/GPU systems. [Figure 3: see original paper] shows the time breakdown for each part of Algorithm 1. Since the time distribution across different problem sizes is similar, we use k=2048 as an example, with x-axis values m(n) representing matrix dimensions.

We find that the GPU kernel occupies most of the time (>70%), while the remaining three data transfer parts account for less than 30% of runtime. Intuitively, data transfer time can be hidden by kernel computation time. To identify operations that can execute in parallel, we categorize resources used in DGEMM into three parts: GPU, CPU+memory bus, and PCIe bus. Operations using different resources can execute concurrently.

shows resource allocation for each step in Algorithm 1. Load1 and store occupy CPU+main memory bus, while load2 only requires PCIe bus for data

transfer. The mult kernel executes on GPU, outputting results to CPU main memory via PCIe bus. Based on resource allocation, using software pipelining to overlap data transfer (load1, load2, store) and mult kernel is feasible. Canqun Yang et al. also implemented pipelining to overlap load1 and mult execution [21]. However, as they noted and our experimental results in Section 4 show, simple pipelining provides modest performance improvement (~20%). In fact, Algorithm 1' s execution time is dominated by the mult kernel. In previous algorithms, the mult kernel directly writes computation results from registers back to remote storage, where the mult kernel includes both matrix multiplication computation and PCIe bus write-back. Thus, after pipelining, data transfer within the kernel becomes a performance bottleneck because: (1) PCIe bandwidth is smaller than GPU memory/CPU main memory bandwidth; (2) mult write-back data is larger than load2 transfer data. For example, in LINPACK DGEMM, k is much smaller than m and n, so output C matrix size m×n is larger than input matrices A, B size k×(m+n). Moreover, as discussed in the next section, we can further develop data reuse optimization to reduce load2 data transfer frequency. Therefore, we optimize the pipelining algorithm to better overlap floating-point operations and data transfer.

3 Pipeline Algorithms

Software pipelining is a general method for overlapping computation and memory operations. Algorithm 1 has three explicit memory operations (load1, load2, store) transferring data between CPU and GPU, while the multiplication operation (mult) directly outputs results to remote storage. Therefore, pipelining to enable parallel execution of mult and the three memory operations is necessary.

3.1 Double Buffering Optimization

Writing back matrix C' s store operation involves data transfer and minimal floating-point computation (beta × C), consuming some CPU resources. To hide the overhead of writing back matrix C, one approach is pipelining across different work units. For example, while work unit i executes store, work unit i+1' s mult operation can proceed simultaneously. This pipeline has two problems. First, resource conflict exists between load1 and store because both transfer data between application space and CAL remote storage (part of CPU main memory), reducing pipeline efficiency. Second, remote storage space is limited, especially cached remote storage. Since store executes on the CPU portion, it uses cached remote storage space to hold generated C sub-matrices, but cache size limits the number of concurrently executing work units in the pipeline.

A better strategy develops finer-grained pipelining within a work unit. Algorithm 2 shows pseudocode for fine-grained pipelining, further dividing generated C sub-matrices into finer blocks executed in a pipelined fashion. We developed a double buffering algorithm that allocates two cache spaces in cached remote storage. For each iteration of lines 6-9 in Algorithm 2, the store operation writes one cached sub-matrix block C_{i,j} back to application space while

mult computes the next sub-matrix block C_{i,j+1} and writes it to the other cache space. During pipelining, mult and store alternately use these two cache spaces. Since GPU kernels execute asynchronously, mult and store can execute in parallel for each iteration.

Algorithm 2: DGEMM with Double Buffering Optimization

```
Partition: A={A , A , ⋯, A }, B={B , B , ⋯, B_q}, C={C , C , ⋯, C_{p×q}}
Work unit: WU={C =A ×B , C =A ×B , ...}
C_{i,j}: 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,⋯,p×q
   // 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} from 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 the last C_{i,j} from remote memory to application space (also multiply by beta)
11. endfor
```

3.2 Data Reuse Optimization

As shown above, the combined execution time of load1, load2, and store is much smaller than mult execution time (Figure 3), so these overheads can be relatively easily hidden through inter-work-unit pipelining. However, resource conflicts remain: load1 and store conflict on CPU main memory, while load2 and mult conflict on PCIe bus. Resource conflicts cause pipeline stalls, reducing overall DGEMM performance.

Fortunately, we can exploit data reuse between consecutive work units. Using Figure 2 as an example, if we schedule work units in a "snake" order, each pair of consecutive work units shares one input sub-matrix, reducing resource conflict overhead. Exploiting data reuse requires two additional steps: First, a preprocessing step reorders work units, storing results in a queue. We partition matrix C into a collection of strip matrices, named with integers (i=0,1⋯). Each strip matrix is further divided into blocks; when i%2=0, partitioning proceeds bottom-to-top; when i%2=1, top-to-bottom. This "wriggled" partitioning connects the top (or bottom) block of one strip matrix with the top (or bottom)

block of the next strip matrix, storing all blocks sequentially in a queue. Second, we need two flags indicating positions of currently loaded A and B matrix blocks in the queue to avoid loading identical blocks. Algorithm 3 presents the pipelining algorithm combining data reuse optimization.

Algorithm 3: DGEMM with Data Reuse Optimization

```
Partition: A={A , A , ⋯, A }, B={B , B , ⋯, B_q}, C={C , C , ⋯, C_{p×q}}
Work unit: WU={C =A ×B , C =A ×B , ...}
C_{i,j}: the sub-matrices C is partitioned into blocks

1. bind remote memory for sub-matrices A, B, C
   // preprocessing: Allocate workunits in a wriggled way
   // the for-loop is pipelined
2. for each workunit wu_i do // i=1,2,⋯,p×q
   // load1
3.   copy either A_i or B_i from application space into remote memory according to the indic
   // 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} from 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 the last C_{i,j} from remote memory to application space (also multiply by beta)
11. endfor
```

3.3 Data Placement Optimization

In data reuse optimization, sub-matrices of A and B are temporarily stored in GPU local memory. We configure remote storage space for A and B as uncached because: (1) matrix A and B data transfer requires no computation; (2) cached remote storage space is too small to effectively accelerate load1 operations. Since C sub-matrices across all work units are independent with no data reuse, and matrix C as final output won' t be reused, storing matrix C in remote storage rather than GPU local memory seems reasonable and reduces limited GPU memory usage. Therefore, in the three algorithms above, matrix C is stored in cached remote storage to gain higher CPU performance for operations like memory-to-memory copy and multiplication by beta.

The above pipelined execution overlaps data transfer operations (load1, load2, store) with the multiplication kernel (mult), making pipeline execution time essentially determined by mult time. Thus, the current optimization direction is

reducing mult execution time in the pipeline. Since the mult kernel includes data transfer operations, outputting data directly from GPU registers to remote storage, our optimization strategy adds an extra pipeline stage so that outputting data to remote storage can also overlap with GPU kernel computation.

We separate the operation of directly outputting matrix C to remote storage from the mult kernel. The kernel outputs results to GPU local memory instead of remote storage. As shown in Algorithm 4, the original mult operation splits into two stages: mult1 and store1, where store1 transfers matrix C from GPU local memory to remote storage. Resource allocation changes: mult1 executes entirely on GPU device, while store1 transfers data via DMA engine. Since both kernel and DMA operations execute asynchronously, we adopt a double buffering strategy in GPU memory to store C sub-blocks, enabling these two operations to execute in parallel. For clarity, we use store2 to refer to the original store operation hereafter.

The optimized DGEMM now establishes a 5-stage pipeline (load1, load2, mult1, store1, store2) with resource allocation shown in Table 2. This successfully addresses the two problems mentioned in Section 2 regarding ACML-GPU. The mult1 kernel requires only GPU resources, no longer needing PCIe bus and system memory, eliminating PCIe contention in mult and enabling parallel execution with load2. Algorithm 4 not only proposes faster kernel execution but also refines pipelining, reducing pipeline stalls caused by resource contention.

Algorithm 4: DGEMM with Data Placement Optimization

```
Partition: A={A , A , ⋯, A }, B={B , B , ⋯, B_q}, C={C , C , ⋯, C_{p×q}}
Work unit: WU={C =A ×B , C =A ×B , ...}
C_{i,j}: the sub-matrices C is partitioned into blocks

1. bind remote memory for sub-matrices A, B, C
   // preprocessing: Allocate workunits in a wriggled way
   // the for-loop is pipelined
2. for each workunit wu_i do // i=1,2,⋯,p×q
   // load1
3.   copy either A_i or B_i from application space into remote memory according to the indic
   // 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} from remote memory to application space (also multiply by beta)
      // mult1
8.     DMAPipeline(C_{i,j})
9.   endfor
   // store2
10.  copy the last C_{i,j} from 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

1. calculate C_{i,j,1} in local memory
2. for each sub-block C_{i,j,k} do // k=2,3···
   // store1
3.   DMA transfer C_{i,j,k-1} from local memory to remote memory
   // mult1
4.   calculate C_{i,j,k} in local memory
5. endfor
   // store1
6. DMA transfer the last C_{i,j,k} from local memory to remote memory
```

[Figure 4: see original paper] shows a coarse-grained space-time diagram of the pipeline, illustrating only the general flow without considering minor resource conflicts between steps. Different shading patterns represent the five operations. Small striped blocks within mult1 indicate sub-matrix data transfer operations overlapped by the mult1 kernel. As shown, except for pipeline startup and termination overhead, most data transfer processes in Algorithm 4 are completely overlapped.

4 Experiments

4.1 Experimental Setup

Our experimental platform is a heterogeneous system with 2-way Intel Xeon 5650 CPU and an ATI HD5970 GPU card. Table 3 summarizes platform configuration parameters. The Intel multi-core CPU delivers 128 GFLOP/s double-precision peak performance. CPU memory size is 24GB with 31GB/s bandwidth. The GPU integrates two Cypress chips with 928 GFLOP/s double-precision peak performance. GPU memory size is 2GB with 256GB/s bandwidth. The heterogeneous system achieves 1056 GFLOP/s double-precision peak performance.

Table 3: System Configuration Parameters | Component | Intel Xeon X5650 Westmere EP 2.66GHz | ATI HD5970 Cypress 725MHz | |———-| ———————-|————–| | Double-precision peak | 128 GFLOP/s | 928 GFLOP/s | | Memory | DDR3 1.3GHz 31.2GB/s | GDDR5 1.0GHz 256GB/s | | Interconnect | PCIe 2.0 x16, 8GB/s | | | Software | icc + openmpi | ATI Stream SDK 2.2 |

For complete experimental analysis, we sequentially run the three optimization algorithms from Section 3, where each optimization includes the previous one. For clarity, we define notation for different optimization versions: * **DB**: Executes Algorithm 2–uses double buffering to hide overhead of writing C matrix back from remote storage to application space. Allocated cache sizes are de-

termined by matrix dimensions in GPU memory. * **DR**: Executes pipelining Algorithm 3, building on DB optimization with improved loading of input matrices, importantly exploiting data reuse for read matrices. * **DP**: Builds on DB and DR, executes Algorithm 4 optimizing data placement in CAL system memory hierarchy. Main optimization changes matrix C storage location and uses DMA for more efficient pipelining of write-back operations. * **HB**: While the above three programs use only GPU computing resources, this program implements hybrid DGEMM—CPU and GPU perform matrix multiplication in parallel. We adopt the load balancing strategy between CPU and GPU from [21], partitioning matrices between CPU and GPU. Experiments launch two MPI processes, each using one CPU/GPU pair.

These four programs represent four optimization strategies with inclusion relationships: DB < DR < DP < HB. Since ACML-GPU [7] library DGEMM code is open source, we use it as our initial baseline and for comparison to measure our optimization effectiveness.

Table 4 shows matrix dimensions (m, n, k) used in experiments. Since different m, n values have minimal impact on DGEMM performance, we set m=n. The k dimension determines data reuse count when loading matrices A, B, affecting DGEMM performance. We use three k values representing different dataset sizes: k=1536, 2048, 4096. In subsequent sections, we derive three DGEMM performance values for different k sizes by averaging performance across different m (or n) scales with the same k value. For detailed profiling, we default to k=2048, with different matrix scales represented by m(n) values on the x-axis.

Table 4: Matrix Dimensions m, n, k Used in Experiments | k | Matrix Dimensions | |–|————–| | 1536 | m=n=6144, 8192, 10240, 12288, 14336, 16384 | | 2048 | m=n=6144, 8192, 10240, 12288, 14336, 16384 | | 4096 | m=n=6144, 8192, 10240, 12288, 14336, 16384 |

4.2 Experimental Results

First, we present optimized DGEMM performance on the heterogeneous system. The baseline for performance comparison is ACML-GPU v1.1. [Figure 5: see original paper] shows performance improvements for the final optimization version DP and hybrid CPU/GPU DGEMM (HB). Hybrid DGEMM (HB-2GPU) achieves maximum performance of 844 GFLOP/s at matrix dimensions (m, n, k) = (16384, 16384, 4096), with corresponding efficiency of 80%. Optimized DGEMM on GPU (DP-2GPU) reaches maximum performance of 758 GFLOP/s at (16384, 16384, 4096), with 82% efficiency. These results show DP-2GPU is twice ACML-GPU library performance, while hybrid DGEMM further improves performance by 10-20%. Generally, all three programs show performance and efficiency improvements as matrix scale increases. Some anomalies occur (e.g., at m=n=10240) because the problem size is not an integer multiple of optimal matrix block sizes for data transfer and kernel execution. As matrix scale increases, DP' s speedup over ACML-GPU decreases: speedups are 2.9×, 2.1×,

and 1.9× for k=1536, 2048, and 4096 respectively. This occurs because the ratio of data transfer to kernel execution time decreases with problem scale. Since our pipeline optimization targets reducing CPU-GPU data transfer overhead, optimization benefits diminish for larger matrices. We observe that large matrices relatively achieve higher efficiency on GPU than small matrices because small matrices have higher data transfer time proportions. Moreover, larger matrix scales yield faster performance improvements for HB-2GPU because CPU DGEMM performance improves with scale, enhancing overall HB-2GPU performance.

Second, we evaluate the three optimization strategies from Section 3. To isolate system interference (e.g., bandwidth contention, detailed in next section), we run DGEMM on a single GPU chip without assigning computation tasks to CPU; CPU only directs data transfer. [Figure 6: see original paper] shows performance improvements for optimized algorithms: Double Buffering (DB), Data Reuse (DR), and Data Placement (DP). Compared to Algorithm 1, double buffering optimization improves performance by 16% by pipelining store2 within a work unit. Data reuse optimization further improves performance by 18% by pipelining data loading across work units. Finally, data placement optimization significantly improves performance by an additional 74% by optimizing the original mult kernel and pipelining C matrix write-back using DMA engine. On a single Cypress GPU chip, the DP optimization algorithm achieves 408 GFLOP/s performance with 88% efficiency.

The three data transfer operations (load1, load2, store) shown in Figure 3 occupy nearly 30% of total computation time, but this does not include all data transfer—store1 execution time should also be added. Our optimized pipeline separates store1 from the mult kernel. Clearly, this approach better reflects pipelining essence. After counting store1 from the mult kernel into total data transfer time, data transfer accounts for over 40% of total time. [Figure 7: see original paper] shows the percentage of each data transfer operation in total data transfer time. Performance improvements from each optimization strategy in Figure 6 generally align with data transfer time distribution, demonstrating our optimizations fully utilize the pipeline. Additionally, data placement optimization provides greater performance improvement because it also enhances kernel performance. Figure 6 also shows optimized DGEMM performance is relatively insensitive to matrix scale, with stable performance. This property provides a good foundation for scaling optimized DGEMM to multiple CPUs and GPUs. DGEMM performance on a single GPU chip is stable, differing from performance trends on heterogeneous platforms in Figure 5. We will discuss this phenomenon further in the next section.

4.3 Experimental Analysis

CPU math library DGEMM typically achieves over 90% efficiency, while hybrid DGEMM on heterogeneous architectures reaches at most 82% efficiency. This section examines: (1) how much optimization space remains for our pipeline

optimization on heterogeneous architectures; (2) how architectural parameters affect optimized DGEMM performance on multiple CPUs/GPUs.

4.3.1 Performance Gap Among the five pipeline stages, the mult1 kernel determines DGEMM' s maximum achievable performance. Nakasato [15] optimized DGEMM kernel performance, achieving 87% efficiency on HD5870 (single Cypress chip). We adopt Nakasato' s optimized kernel but use image read/write addressing mode instead of global addressing mode, achieving higher efficiency: 94% on a single Cypress chip in our HD5970.

[Figure 8: see original paper] compares DGEMM efficiency with kernel efficiency, including optimized DGEMM on one GPU (DP), two GPUs (DP-2GPU), and hybrid DGEMM with two CPUs and two GPUs (HB-2GPU). We calculate average efficiency for each test set. DGEMM calls the kernel multiple times; we time each kernel invocation and average their performance as final kernel performance. The optimized kernel only reads/writes GPU local memory, so its performance is CPU-independent. As shown, kernel average efficiency exceeds 90% (peak 94%), similar to CPU DGEMM efficiency. The difference between DP and kernel is the presence/absence of CPU-GPU data transfer via memory bus and PCIe bus; our software pipelining masks data transfer overhead. Experimental results show DP-1GPU efficiency drops 6% compared to kernel due to data transfer overhead. This performance degradation has two causes: First, pipeline startup and termination time cannot be hidden, accounting for ~3% of total DGEMM time. Second, as Table 2 shows, inherent resource conflicts remain in optimized DGEMM: memory bus conflict between load1 and store2, PCIe bus conflict between load2 and store1. Excluding these factors, we believe optimized DGEMM on a single GPU chip (DP-1GPU) nearly achieves optimal performance.

[Figure 8: see original paper] also shows efficiency drops when DGEMM runs on more CPUs and GPUs (DP-2GPU and HB-2GPU). When DP runs on two GPU chips (DP-2GPU), efficiency drops 11% compared to DP-1GPU. Both DP-1GPU and DP-2GPU require CPU-GPU data transfer via memory bus and PCIe bus, intensifying resource contention on both buses. Additionally, when DGEMM scales to a heterogeneous system with two CPUs and two GPUs (HB-2GPU), efficiency drops 5% compared to DP-2GPU. In HB-2GPU, CPUs run partial DGEMM computations, sharing application space with GPUs and further burdening the memory bus. Increased system memory contention makes HB-2GPU efficiency lower than DP-2GPU. We discuss these two resource contention issues in detail in the following sections.

4.3.2 Scalability on Multiple GPUs Most accelerators (e.g., GPU, ClearSpeed, Tilera) connect to CPU via PCIe bus; multiple PCIe slots on motherboards can support multiple GPUs simultaneously. Some GPU cards integrate multiple GPU chips, such as ATI Radeon HD5970 and NVIDIA Tesla S1070. Therefore, optimized DGEMM scalability on multiple GPUs is also important.

Due to platform limitations, experiments run two MPI processes, each responsible for one GPU chip. Since shared resource contention (PCIe bus and memory bus) is the key factor affecting DGEMM scalability, using two GPU chips' performance to predict DGEMM performance on multiple GPUs is feasible. Experiments attempt to predict DGEMM scalability on multiple GPUs through bandwidth contention between two GPU chips. Experiments profile effective bandwidth changes for DGEMM from one to two GPU chips, where each MPI process in DP-2GPU runs the same matrix scale as DP-1GPU. As Table 2 shows, both PCIe bus and memory bus experience bandwidth contention.

To highlight bandwidth changes, we normalize PCIe and memory bandwidth in DP-2GPU relative to DP-1GPU bandwidth. First, we consider PCIe bandwidth contention between load2 and store1. [Figure 9: see original paper] shows average bandwidth reduction, with y-axis representing normalized relative bandwidth. As shown, effective bandwidth for load2 and store1 reaches 89% and 56% of DP-1GPU respectively. Since store1 transfers more frequently and larger data 规模 (matrix C is larger than matrices A and B combined), its bandwidth drops more significantly. As mentioned in Section 3.3, to fully utilize pipelining, the mult1 kernel further partitions sub-matrices into smaller blocks. Each store1 operation handles a smaller matrix sub-block. During mult1 execution, multiple store1 operations execute simultaneously (in our experiments, 4 store1 operations execute concurrently with 1 mult1, depending on kernel sub-block size). During mult1 execution time, PCIe bandwidth can be considered occupied by store1, so even DGEMM on a single GPU chip already has high store1 PCIe bandwidth occupancy. Therefore, when scaling to two GPU chips, PCIe bus contention becomes more severe. However, load2' s PCIe bandwidth degradation is less severe than store1 because load2 pipelines across work units with mult1 kernel, unlike store1' s intra-work-unit pipelining with mult1, making PCIe requests less frequent. Additionally, load2 transfers matrix 规模 $(m+n)\times k$, while store1 transfers $m\times n$. Since k is much smaller than n, the former exerts less pressure on PCIe bus.

Second, besides PCIe bandwidth contention, memory bus contention exists between two GPU chips because data transfer also occurs between application space and CAL remote storage. [Figure 10: see original paper] shows effective memory bandwidth for load1 and store2 drops by 8% and 14% respectively. Similar to PCIe bandwidth degradation, store2' s bandwidth drop is more pronounced.

Through pipeline execution analysis, we find that in DP-1GPU, load1, load2, store1, and store2 are almost completely hidden by mult1. However, [Figure 8: see original paper] shows effective bandwidth reduction still causes 11% efficiency drop in DP-2GPU, indicating shared resource contention (PCIe bandwidth and memory bandwidth) prevents some data transfer from overlapping with mult1 kernel. As GPU count increases, bandwidth requests become more frequent, intensifying bandwidth contention' s overall performance impact. In summary, when DGEMM scales from 1 to 2 GPU chips, both PCIe and system

memory bandwidth decrease. Based on these results, we draw two conclusions: * **Conclusion 1:** DGEMM scalability across multiple GPU cards on the same motherboard is limited by PCIe bandwidth. In DGEMM, both load2 and store1 use PCIe bandwidth; as Figure 7 shows, these two operations account for 60% of total DGEMM data transfer time. Experimental results in Figure 8 demonstrate contention-caused efficiency reduction on two GPU chips (DP-2GPU). As GPU count increases, PCIe contention intensifies, further impacting DGEMM efficiency. * **Conclusion 2:** Improving system memory bandwidth enhances GPU-only DGEMM (DP-1GPU and DP-2GPU) performance. Although load1 and store2 both consume memory bandwidth, Figure 10 shows they are less sensitive to memory bandwidth contention. As Figure 7 shows, these portions are not the main components of data transfer time. While pinned memory usage could avoid load1 and store2 contention in some cases, it requires data not be reallocated and data 规模 smaller than pinned memory limits. Our work proves this transfer overhead can also be reduced through algorithmic optimization.

4.3.3 Scalability on Hybrid CPUs and GPUs On our hybrid experimental platform, Intel Xeon CPU provides 128 GFLOP/s computing capability, accounting for 12% of system double-precision floating-point performance. For compute-intensive programs like DGEMM, CPU computing capability cannot be ignored. In hybrid DGEMM HB-2GPU implementation, matrices are first divided into two equal parts, each computed by one CPU/GPU pair. Within each CPU/GPU pair, we adopt the partitioning algorithm from [21] to divide tasks between CPU and GPU. Although HB-2GPU improves performance by 6% over DP-2GPU, efficiency drops by 5%. This section analyzes efficiency degradation causes.

[Figure 8: see original paper] shows DGEMM efficiency on hybrid CPUs/GPUs system. [Figure 11: see original paper] analyzes CPU portion (CPU-HB) contribution to HB-2GPU performance. Pure CPU DGEMM (Pure-CPU) serves as comparison, computing the same matrix scale as CPU-HB. The figure shows hybrid DGEMM CPU portion performance is 22% lower than Pure-CPU because GPU-computed DGEMM in hybrid version also copies data from application space to CAL remote storage, interfering with CPU DGEMM computation and intensifying CPU memory contention. We conclude: **Conclusion 3:** Improving system memory bandwidth helps reduce memory contention, thereby improving hybrid DGEMM performance on CPU/GPU heterogeneous systems. As CPU computing capability increases, system memory contention' s impact on hybrid DGEMM overall performance grows. Pinned memory usage would help reduce memory contention.

Another efficiency degradation cause is load imbalance between CPU and GPU. We use a heuristic partitioning strategy keeping CPU and GPU execution time difference within a threshold. Based on multiple experiments, this paper uses 0.1 seconds as threshold. [Figure 12: see original paper] plots relative difference between CPU and GPU execution time, with y-axis ((GPU time - CPU time) /

CPU time). This slight imbalance causes minor overall performance reduction (~5%).

5 Related Work

Previous works have optimized DGEMM on GPUs for matrix scales smaller than GPU memory. Nakasato proposed a new DGEMM kernel [15] based on HD5870, achieving 87% of GPU peak performance. Our optimized DGEMM kernel reaches 94% efficiency on a single Cypress chip of HD5970. Chris Jang' s GATLAS auto-tuner [20] uses auto-tuning to enhance portability across different GPU architectures, intended for invocation in real applications. GATLAS also only considers cases where matrices fit in GPU memory, thus currently having no direct way for real application invocation. V. Volkov and J. Demmel implemented one-sided matrix factorizations (LU, QR, etc.) on hybrid CPU/GPU systems [12], distributing factorization across CPU and GPU for simultaneous execution. However, their matrix multiplication still only handles matrices that fit in GPU memory, without CPU-GPU data transfer. Many other works target GPU performance optimization for specific programs, none considering data transfer, which we won' t enumerate here.

Some works consider data transfer overhead but mostly focus on parallelizing CPU and GPU computation without reducing data transfer overhead. S. Venkatasubramanian and R. Vuduc implemented Jacobi algorithm on hybrid CPU/GPU architecture [23]. They considered CPU-GPU data transfer, but hybrid program performance improvement was only 8%. DaQi Ren and Reiji Suda implemented large-scale matrix multiplication on multi-core CPU/GPU systems [25]. They focused on energy consumption, using CPU multi-threading for optimization. When one thread waits for GPU memory access completion signals, CPU launches a new thread for other tasks. This method enables simultaneous GPU and CPU computation, but data transfer still stalls entire execution, wasting CPU and GPU computing capability. C. Feichtinger et al. implemented parallel Lattice Boltzmann method on hybrid CPU/GPU clusters [24]. They minimized transferred data by only transmitting PDE boundary values, identifying load imbalance as one reason for limited performance improvement. Y. Ogata et al. proposed a model-based heterogeneous FFT library on CPU/GPU systems [27]. This model better guides task partitioning between CPU and GPU. We adopt the adaptive partitioning algorithm from [21] when optimizing DGEMM on heterogeneous systems. Our work focuses on large-scale DGEMM implementation on heterogeneous CPU/GPU architectures, including CPU-GPU data transfer. We not only count data transfer in total time but also optimize it with pipelining algorithms, enabling this overhead to overlap with computation. Consequently, hybrid DGEMM can be invoked in real applications. Through our optimization, hybrid DGEMM achieves 844 GFLOP/s performance with 80% efficiency. Canqun Yang et al. proposed masking data transfer overhead with DGEMM computation [21], using optimizations like data loading pipelining and data output pipelining. We further developed data place-

ment optimization strategies to improve DGEMM kernel performance and established finer-grained pipelining. This additional optimization strategy improves DGEMM performance by 74%, becoming the most important optimization. Additionally, we analyze shared resource contention (particularly PCIe bus and system memory) and optimized DGEMM scalability across multiple CPUs and GPUs. Through analysis, we provide recommendations for scaling DGEMM to heterogeneous CPUs/GPUs architectures.

6 Conclusion

We optimized large-scale DGEMM through three strategies (double buffering, data reuse, and data placement), developing a new pipelining algorithm. In the pipeline, we overlap DGEMM kernel execution on GPU with data transfer processes. Optimized DGEMM achieves 408 GFLOP/s (88% efficiency) on a single ATI HD5970 Cypress chip, 758 GFLOP/s (82% efficiency) on ATI HD5970, and 844 GFLOP/s (80% of peak) for hybrid DGEMM on heterogeneous CPU/ATI GPU systems. Comparison with kernel performance shows optimized DGEMM on a single GPU chip approaches peak performance with limited further optimization space. However, when scaling DGEMM to multiple GPUs and CPUs, efficiency is impacted, primarily by shared resource contention, particularly PCIe and system memory contention. Through experiments and analysis, we draw three conclusions: (1) DGEMM scalability across multiple GPU cards on the same motherboard is limited by PCIe bandwidth; (2) Improving system memory bandwidth enhances GPU-only DGEMM (DP-1GPU and DP-2GPU) performance; (3) Improving system memory bandwidth helps reduce system memory contention, thereby improving hybrid DGEMM performance on CPU/GPU heterogeneous systems.

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 Library for Graphic Processors (ACML-GPU) http://developer.amd.com/gpu/acmlgpu/pages/default

[8] NVIDIA. 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

[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

Author Biographies

Jiajia Li: Ph.D. candidate, Institute of Computing Technology, 2010, lijiajia@ict.ac.cn
Xingjian Li: Master' s student, Institute of Computing Technology, 2008
Guangming Tan: Associate Professor, Institute of Computing Technology

Note: Figure translations are in progress. See original paper for figures.