Performance Study of DGEMM on CPU/ATI GPU Hybrid Architecture Postprint

Authors: Li Jiajia, Li Xingjian, Tan Guangming

Date: 2016-06-08T00:00:00+00:00

Abstract

This paper reports our work on optimizing Double Precision General Matrix Multiplication (DGEMM) on CPU/ATI GPU hybrid architectures. In real-world applications, data transfer between CPU and graphics processor (GPU) is a key factor affecting performance. Since software pipelining can reduce data transfer overhead, we propose three software pipelining algorithms: Double Buffering, Data Reuse, and Data Placement. On AMD' s graphics processor (GPU) ATI HD5970, the optimized DGEMM performance reaches 758 GFLOP/s, corresponding to an efficiency of 82%, which is twice the performance of ACML-GPU v1.1. On a heterogeneous system composed of Intel Westmere EP and ATI HD5970, the performance reaches 844 GFLOP/s with an efficiency of 80%. We further investigate the scalability of DGEMM on multiple CPUs and multiple GPUs, and analyze in detail the architectural impact factors. Analysis shows that contention on the PCIe bus and memory bus is an important factor affecting performance degradation of programs on heterogeneous systems.

Full Text

Preamble

Vol. 9 No. 6 Information Technology Letters Vol.9 No.6 Performance Study of DGEMM on CPU/ATI GPU Hybrid Architecture Jiajia Li, Xingjian Li, Guangming Tan

Abstract

This paper reports our work on optimizing Double-precision GEneral Matrix Multiply (DGEMM) on CPU/ATI GPU hybrid architectures. In real applications, data transfer between CPU and Graphics Processing Unit (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, the performance reaches 844 GFLOP/s with 80% efficiency. We further investigate the scalability of DGEMM on multiple CPUs and multiple GPUs, analyzing architectural influences in detail. Our analysis shows 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 various scientific and engineering applications. Many key numerical algorithms, such as BLAS Level 3 [1] and LU decomposition [2], rely on high-performance implementations of DGEMM. The HPL [3] benchmark, used for ranking supercomputers worldwide, consists of dense matrix LU decomposition. Since DGEMM performance heavily depends on computer hardware, many processor vendors have developed machine-specific DGEMM implementations, such as Intel' s MKL and AMD' s ACML, and further optimized BLAS libraries on their respective multi-core processors.

The floating-point peak performance of Graphics Processing Units (GPUs) is more than an order of magnitude higher than that of general-purpose CPUs. For example, AMD HD5970 achieves 928 GFLOP/s double-precision floating-point performance, while NVIDIA Tesla C2070 has a double-precision floating-point peak performance of 515 GFLOP/s. DGEMM is a compute-intensive program with regular memory access patterns, making it suitable for optimization on GPUs. Indeed, numerous works have optimized DGEMM on GPUs [11-17,19-21,23-27]. Most previous work focused on cases where the computation matrices were already stored in GPU memory, optimizing DGEMM implementation on GPUs. Currently, in many systems, GPUs are connected to CPUs via PCIe bus as accelerator components. In real applications, DGEMM computation 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 DGEMM computation on GPU. 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. The peak memory bandwidth of GPU HD5970 is 256 GB/s, while the peak PCIe bandwidth is 8 GB/s. Although DGEMM is compute-intensive, this bandwidth difference remains a bottleneck

for overall DGEMM performance on heterogeneous systems. N. Nakasato noted in reference [15] that if data transfer time between CPU and ATI Cypress GPU is included in DGEMM' s total time, DGEMM efficiency drops from 85% to 55%. AMD' s GEMM optimization library for CPU-ATI GPU heterogeneous systems also exhibits this efficiency degradation.

Optimizing DGEMM and analyzing future hybrid architecture trends require quantitative analysis of each memory hierarchy level: understanding how much impact the current CPU-GPU heterogeneous architecture and memory hierarchy have on DGEMM' s overall performance. Although previous works have optimized DGEMM on GPUs, few articles provide quantitative analysis, especially for CPU/GPU heterogeneous systems.

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

The main contributions of this paper are threefold:

- We quantitatively analyze DGEMM performance on heterogeneous architectures, with detailed analysis of the CPU-GPU data transfer process. The analysis shows that data transfer accounts for 40% of DGEMM' s total time, rather than the 20% reported in previous work, indicating that data transfer overhead significantly impacts DGEMM performance.
- We propose new 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 on a single Cypress GPU, corresponding to 88% efficiency. The optimized DGEMM performance is more than twice that of ACML-GPU v1.1. On ATI HD5970, the optimized DGEMM reaches 758 GFLOP/s with 82% efficiency. On a heterogeneous system with Intel Westmere EP and ATI HD5970, the performance reaches 844 GFLOP/s, achieving 80% of peak performance.
- Finally, we analyze resource contention when scaling DGEMM to multiple CPUs and multiple GPUs, focusing on PCIe contention and memory contention. Our analysis shows that resource contention (PCIe and memory) is a primary bottleneck for scalability on heterogeneous systems. However, software approaches alone cannot completely avoid the impact of resource contention on DGEMM' s overall performance.

2 Background

Our performance optimization primarily targets the ATI Evergreen GPU architecture; Cypress is the most advanced GPU in the Evergreen series. The ATI HD5870 card encapsulates one Cypress chip, while HD5970 integrates two Cypress chips. This section introduces several important characteristics of Cypress —the memory hierarchy at the ATI CAL [6] software level. We use the DGEMM program from the ACML-GPU library as our baseline and foundation for per-

formance optimization. Therefore, we conduct a detailed analysis of DGEMM in ACML-GPU 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 fully exploit floating-point operation throughput, the Cypress GPU microarchitecture employs Single Instruction Multiple Data (SIMD) and Very Long Instruction Word (VLIW). Since optimizing DGEMM' s kernel [5] on GPU is not the focus of this paper, we do not describe GPU microarchitecture details here. We use the kernel from reference [15] with 80% efficiency. Briefly, a Cypress card at 725MHz achieves 2.32 TFLOP/s single-precision floating-point peak performance, with double-precision floating-point operation performance being 1/5 of single-precision performance, thus delivering 464 GFLOP/s double-precision floating-point peak performance at 725MHz.

Figure 1 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 system divides physical memory into local memory and remote memory. Local memory refers to GPU memory, the high-speed memory on the graphics card. Remote memory refers to memory space not physically on the graphics card but accessible by GPU (i.e., part of CPU main memory). Both local and remote memory can be read by GPU kernels. In other words, GPU kernels can write data from GPU registers back to either GPU memory or remote memory. Since operations on remote memory have longer latency than those on local memory, this memory approach has lower performance. Remote memory is further divided into cached and uncached portions. For example, CAL system divides HD5970' s remote memory into 1788MB of uncached memory and 500MB of cached memory. Therefore, choosing which memory space to use as shared data space 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 memory, and between remote memory and local memory. One optimization method uses system pinned memory—applications directly transfer data from CPU application space to remote memory space for DMA transfer. This approach eliminates data transfer from application space to remote memory. Some applications implemented using CUDA [5] effectively improve performance through this method. CAL systems also have pinned memory technology, but with limitations on size and other aspects. Therefore, reducing PCIe bus data transfer time through proper organization between memory hierarchy levels (such as pipeline algorithms) is essential.

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 $C \leftarrow \alpha A \times B + \beta C$, where A, B, and C are matrices of dimensions $m \times k$, $k \times n$, and $m \times n$, respectively. In practical applications involving DGEMM, these three matrices are large and cannot all be stored in GPU memory. Therefore, these matrices are partitioned into multiple sub-matrix blocks, with several matrix blocks transferred to GPU memory at a time for partial DGEMM computation. The partitioning algorithm divides A, B, and C into $\{A_1, A_2, ..., A_p\}$, $\{B_1, B_2, ..., B_q\}$, and $\{C_1, C_2, ..., C_{p\times q}\}$, where p and q depend on GPU memory size. Using $p = 2$, $q = 2$ as an example, Figure 2 shows the partitioning creates four independent C sub-matrices: $C_1 = A_1 \times B_1$, $C_2 = A_1 \times B_2$, $C_3 = A_2 \times B_1$, $C_4 = A_2 \times B_2$. 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, and the DGEMM kernel further divides these sub-matrices into smaller matrices [9-13] (such as 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 memory) must go through remote memory (line 1). In Algorithm 1 pseudocode, loading data to GPU includes two steps (load1 and load2), while writing data back to CPU memory is one step (store). In fact, the DGEMM kernel multiplication operation on line 5 implicitly includes another data write-back operation—sub-matrix C data is written back from GPU registers to remote memory.

Algorithm 1. Original DGEMM Implementation

```
Partition: A={A1, A2, ⋯, Ap}, B={B1, B2, ⋯, Bq}, C={C1, C2, ⋯, Cpxq}
Work unit: WU={C1=A1*B1, C2=A1*B2, ...}

1. bind remote memory for sub-matrices A, B, C
2. for each workunit wui do // i=1, 2, ⋯, p×q
   // load1
3.   copy both Ai and Bi from application space into remote memory
   // load2
4.   copy both Ai and Bi from remote memory to local memory
   // mult
5.   calculate Ci on GPU device and directly output it to remote memory
   // store
6.   copy Ci from remote memory to application space (also multiply by beta)
7. endfor
```

We use Algorithm 1 as our baseline program for performance comparison. Below we provide a detailed breakdown of DGEMM runtime components on hetero-

geneous CPU/GPU systems. Figure 3 shows the time for each component of Algorithm 1. Since the time distribution of DGEMM components is similar across different problem sizes, we use $k = 2048$ as an example, with $m(n)$ values on the x-axis representing matrix dimensions.

We find that the GPU kernel multiplication occupies most of the time (over 70%), while the remaining three data transfer components 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.

Table 1. Resource Allocation in Algorithm 1

PCIe Bus	CPU+Memory Bus	GPU
load1, load2	store	mult

Table 1 shows resource usage for each component 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 and outputs 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 stated and our experimental results in Section 4 show, simple pipelining algorithms provide limited performance improvement (about 20%). In fact, the execution time of Algorithm 1 is dominated by the mult kernel. In previous algorithms, the mult kernel directly writes computation results from registers back to remote memory, where the mult kernel includes both matrix multiplication computation and data write-back via PCIe bus. Consequently, after pipelining, data transfer in the kernel becomes a performance bottleneck because: (1) PCIe bandwidth is smaller than GPU memory/CPU main memory bandwidth; (2) Data written back by mult is larger than data transferred in load2 operations. For example, in LINPACK' s DGEMM, k is much smaller than m and n, so the output C matrix size $m \times n$ is larger than input matrices A and B sizes $k \times (m + n)$. Moreover, as discussed in the next section, we can further reduce data transfer times in load2 through data reuse optimization. Therefore, we optimize the pipelining algorithm to better overlap floating-point operations with data transfer operations.

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 memory. Therefore, pipelining

algorithms are necessary to enable parallel execution of mult and the three memory operations.

3.1 Double Buffering Optimization

Writing back the C matrix via store operations involves data transfer and a small amount of floating-point computation ($\beta \times C$), consuming certain CPU resources. To hide the overhead of writing back the C matrix, one approach is to pipeline across different work units. For example, while work unit i executes store operations, work unit $i+1$ can execute mult operations simultaneously. This pipeline has two problems. First, resource conflict exists between load1 and store because both transfer data between application space and CAL remote memory (part of CPU main memory), reducing pipeline efficiency. Second, remote memory space is limited, especially cached remote memory. Since store operations execute on the CPU side, cached remote memory space is used 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 single work unit. Algorithm 2 shows pseudocode for fine-grained pipelining, where generated C sub-matrix blocks are further divided into smaller sub-blocks executed in a pipelined manner. We developed a double buffering algorithm that creates two buffer spaces in cached remote memory. For each iteration of the for-loop in lines 6-9 of Algorithm 2, the store operation writes one buffered 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 buffer space. During pipelining, mult and store alternately use these two buffer 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={A1, A2, ⋯, Ap}, B={B1, B2, ⋯, Bq}, C={C1, C2, ⋯, Cpxq}
Work unit: WU={C1=A1*B1, C2=A1*B2, ...}
Cij: the sub-matrices C is partitioned into blocks

1. bind remote memory for sub-matrices A, B, C
2. for each workunit wui do // i=1, 2, ⋯, p×q
   // load1
3.   copy both Ai and Bi from application space into remote memory
   // load2
4.   copy both Ai and Bi from remote memory to local memory
   // mult
5.   calculate Ci,1 on GPU device and output it to remote memory
6.   for each block Ci,j do // j=2, 3, ⋯
      // store
7.     copy Ci,j-1 from remote memory to application space (also multiply by beta)
      // mult
```

```
8.     calculate Ci,j on GPU device and output it to remote memory
9.   endfor
   // store
10.  copy the last Ci,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 operations is much smaller than mult execution time (Figure 3), making their overhead relatively easy to hide through inter-work-unit pipelining. However, resource conflicts still exist between operations: 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 $\times$ 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, with results stored in a queue. We partition the C matrix 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-up; when $i\%2 = 1$, top-down. We partition strip matrices in this "wriggled" way, where blocks at the top (or bottom) of one strip matrix connect to blocks at the top (or bottom) of the next strip matrix, and all blocks are stored sequentially in the queue. Second, we need two flags to indicate the positions of currently running A and B matrix blocks in the queue to avoid loading identical matrix blocks. Algorithm 3 shows the pipelining algorithm combined with data reuse optimization.

Algorithm 3. DGEMM with Data Reuse Optimization

```
Partition: A={A1, A2, ⋯, Ap}, B={B1, B2, ⋯, Bq}, C={C1, C2, ⋯, Cpxq}
Work unit: WU={C1=A1*B1, C2=A1*B2, ...}
Cij: the sub-matrices C is partitioned into blocks

1. bind remote memory for sub-matrices A, B, C
   // pre-processing: Allocate workunits in a wriggled way
   // the for-loop is pipelined
2. for each workunit wui do // i=1, 2, ⋯, p×q
   // load1
3.   copy either Ai or Bi from application space into remote memory according to the indicat
   // load2
4.   copy either Ai or Bi from remote memory to local memory according to the indicators
   // mult
5.   calculate Ci,1 on GPU device and output it to remote memory
6.   for each block Ci,j do // j=2, 3, ⋯
```

```
      // store
7.     copy Ci,j-1 from remote memory to application space (also multiply by beta)
      // mult
8.     calculate Ci,j on GPU device and output it to remote memory
9.   endfor
   // store
10.  copy the last Ci,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 memory space for A and B as uncached because: (1) No computation is needed during data transfer of matrices A and B; (2) The cached portion of remote memory is too small to effectively accelerate load1 operations. Since no data reuse exists between C sub-matrices across all work units (they are independent) and the C matrix is the final output not reused, storing the C matrix in remote memory rather than GPU local memory seems reasonable and reduces occupancy of limited GPU memory. Therefore, in the three algorithms above, the C matrix is stored in cached remote memory to achieve higher CPU performance for operations like memory-to-memory copies and multiplication by β.

The above pipelined execution overlaps data transfer operations (load1, load2, store) with the multiplication kernel (mult), with 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 memory, our optimization strategy adds an additional pipeline stage so that outputting data to remote memory can also overlap with GPU kernel computation.

We separate the operation of directly outputting matrix C to remote memory from the mult kernel. The kernel outputs results to GPU local memory instead of remote memory. As shown in Algorithm 4, the previous mult operation is split into two stages: mult1 and store1, where store1 transfers matrix C from GPU local memory to remote memory. Resource usage changes: mult1 executes entirely on GPU device, while store1 transfers data via DMA engine. Since both kernel and DMA operations execute asynchronously, we also 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 store operation hereafter.

The optimized DGEMM now establishes a 5-stage pipeline (load1, load2, mult1, store1, store2), with resource usage for each operation shown in Table 2. We have successfully solved the two problems in ACML-GPU mentioned in Section 2. The mult1 kernel only requires 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 the pipeline, reducing pipeline stalls caused by resource contention.

We sketch a space-time diagram of the pipeline (Figure 4). This diagram only shows the general pipeline flow without considering resource conflicts between micro-steps. We use different shading patterns for the five operations. The striped small blocks in the mult1 segment represent data transfer operations of sub-matrices overlapped by the mult1 kernel. As shown, except for pipeline startup and termination overhead, most data transfer processes in Algorithm 4 are completely overlapped.

Table 2. Resource Allocation in Optimized DGEMM (Algorithm 4)

PCIe Bus	CPU+Memory Bus	GPU
load2, store1	load1, store2	mult1

Algorithm 4. DGEMM with Data Placement Optimization

```
Partition: A={A1, A2, ⋯, Ap}, B={B1, B2, ⋯, Bq}, C={C1, C2, ⋯, Cpxq}
Work unit: WU={C1=A1*B1, C2=A1*B2, ...}
Cij: the sub-matrices C is partitioned into blocks

1. bind remote memory for sub-matrices A, B, C
   // pre-processing: Allocate workunits in a wriggled way
   // the for-loop is pipelined
2. for each workunit wui do // i=1, 2, ⋯, p×q
   // load1
3.   copy either Ai or Bi from application space into remote memory according to the indicat
   // load2
4.   copy either Ai or Bi from remote memory to local memory according to the indicators
   // mult1
5.   DMAPipeline(Ci,1)
6.   for each block Ci,j do // j=2, 3, ⋯
      // store2
7.     copy Ci,j-1 from remote memory to application space (also multiply by beta)
      // mult1
8.     DMAPipeline(Ci,j)
9.   endfor
   // store2
10.  copy the last Ci,j from remote memory to application space (also multiply by beta)
11. endfor

Algorithm: DMAPipeline(Ci,j)
Ci,j,k: the Ci,j blocks are further partitioned into sub-blocks

1. calculate Ci,j,1 in local memory
```

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

4.1 Experimental Setup

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

Table 3. System Configuration Parameters

Component	Intel Xeon X5650	ATI HD5970
Architecture	Westmere EP 2.66GHz	Cypress 725MHz
Double-precision peak	128 GFLOP/s	928 GFLOP/s
Memory	DDR3 1.3GHz	GDDR5 1.0GHz
Memory bandwidth	31.2 GB/s	256 GB/s
Interconnect	PCIe 2.0 x16, 8GB/s	
Software	icc + openmpi	ATI Stream SDK 2.2

To provide 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 optimized versions:

- **DB**: This program executes Algorithm 2—using double buffering to hide overhead of writing C matrix back from remote memory to application space. The two allocated buffer sizes are determined by matrix dimensions in GPU memory.
- **DR**: This program executes pipelining Algorithm 3, improving upon DB by reducing input matrix loading overhead, with an important optimization being data reuse for loaded matrices.
- **DP**: Building on DB and DR, this program executes Algorithm 4, optimizing data placement in CAL system memory hierarchy. The main optimization changes matrix C storage location and designs a more effective pipeline using DMA.

- **HB**: While the above three programs utilize 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 reference [21], partitioning matrices between CPU and GPU. In experiments, we launch two MPI processes, each using a CPU/GPU pair.

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

Table 4 shows matrix dimensions (m, n, k) used in experiments. Since different m and n values have minimal impact on DGEMM performance, we set $m = n$. The k dimension determines data reuse count when loading input matrices A and B, affecting DGEMM performance. We use three k values representing different dataset sizes: $k = 1536$, 2048, and 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 breakdowns, 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	$m = n$
1536	6144, 8192, 10240, 12288, 14336, 16384
2048	6144, 8192, 10240, 12288, 14336, 16384
4096	6144, 8192, 10240, 12288, 14336, 16384

4.2 Experimental Results

First, we present optimized DGEMM performance on heterogeneous systems. The baseline for performance comparison is ACML-GPU v1.1. Figure 5 plots performance improvements for the final optimized version DP and hybrid DGEMM (HB) with CPU/GPU 协同计算. 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 $(m, n, k) = (16384, 16384, 4096)$, with 82% efficiency. These results show DP-2GPU is twice ACML-GPU library DGEMM performance, and hybrid DGEMM further improves performance by 10%-20%. Generally, all three programs show performance and efficiency improvements as matrix scale increases. Some scales exhibit anomalies (when $m = n = 10240$) because the problem size is not an integer multiple of optimal data transfer and kernel execution block sizes. As matrix scale increases, DP' s speedup over ACML-GPU decreases. For $k = 1536$, 2048, and 4096, speedups are 2.9×, 2.1×, and

1.9× respectively. This is because the ratio of data transfer to kernel execution time decreases as problem size increases. Since our pipeline optimization targets reducing CPU-GPU data transfer overhead, optimization effectiveness diminishes for larger matrices. We observe that large-scale matrices relatively achieve higher efficiency on GPU than small-scale matrices because small matrices have larger data transfer time proportions. And as matrix scale increases, HB-2GPU performance improves faster because CPU-side DGEMM performance improves with larger matrices, enhancing overall HB-2GPU performance.

Second, we evaluate the three optimization strategies proposed in Section 3. To shield from system interference (such as bandwidth contention, detailed in the next section), we run DGEMM on a single GPU chip without assigning computation tasks to CPU; CPU only directs data transfer. Figure 6 shows performance improvements for the optimized algorithms: double buffering (DB), data reuse (DR), and data placement (DP). Compared to Algorithm 1, double buffering 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 another 74%, where we optimize the original mult kernel and pipeline C matrix write-back using DMA engine. On a single Cypress GPU chip, the DP optimization algorithm reaches 408 GFLOP/s with 88% efficiency.

The three data transfer operations shown in Figure 3 (load1, load2, and store) occupy nearly 30% of total computation time, but this is not all data transfer time—store1 execution time should also be included. We optimize the pipeline algorithm by separating store1 from the mult kernel. Obviously, this approach better matches the essence of pipelining. After including store1 from the mult kernel in total data transfer time, data transfer processes account for over 40% of total time. Figure 7 shows the percentage of each data transfer operation in total data transfer time. The performance improvements from each optimization strategy in Figure 6 generally align with the distribution of data transfer time components, demonstrating that our optimizations fully utilize the pipeline. Additionally, data placement optimization provides greater performance improvement because it also enhances kernel performance. Figure 6 also shows that optimized DGEMM performance is relatively insensitive to matrix scale changes, with stable performance. This property provides a good foundation for scaling optimized DGEMM to multiple CPU and multiple GPU platforms. The stable DGEMM performance on a single GPU chip in this figure differs from performance trends on heterogeneous platforms in Figure 5, which we will discuss further in the next section.

4.3 Experimental Analysis

DGEMM efficiency in CPU math libraries typically exceeds 90%, while hybrid DGEMM efficiency on heterogeneous architectures reaches at most 82%. 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 CPU/GPU systems.

4.3.1 Performance Gap

Among the five pipeline stages, the mult1 kernel determines DGEMM' s maximum achievable performance. N. Nakasato optimized DGEMM kernel performance, achieving 87% efficiency on HD5870 (integrating one Cypress chip) [15]. We adopt Nakasato' s optimized kernel but use image read/write addressing mode instead of global addressing mode, achieving higher efficiency on a Cypress chip.

Figure 8 compares DGEMM efficiency with kernel efficiency, including optimized DGEMM on one GPU (DP implementation), DP on two GPUs, and hybrid DGEMM on two CPUs and two GPUs. We calculate average efficiency for each test set. DGEMM calls the kernel multiple times during execution; we time each kernel call 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% (optimal efficiency is 94%), similar to CPU DGEMM efficiency. The difference between DP and kernel is the presence or absence of data transfer between CPU and GPU via memory bus and PCIe bus, which we hide using software pipelining. Results show DP-1GPU efficiency drops 6% compared to the kernel due to data transfer overhead. This performance degradation has two causes: First, pipeline startup and termination time cannot be hidden, accounting for about 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 achieves near-optimal performance on a single GPU chip (DP-1GPU).

Figure 8 also shows efficiency degradation 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 data transfer between CPU and GPU via memory bus and PCIe bus, intensifying contention on these two 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, CPU runs partial DGEMM, sharing application space with GPU and further burdening the memory bus. Increased system memory contention makes HB-2GPU efficiency lower than DP-2GPU. We will discuss these two resource contention issues in detail in the following sections.

4.3.2 Scalability on Multiple GPUs

Currently, most accelerators (e.g., GPU, ClearSpeed, Tilera) connect to CPU via PCIe bus, with multiple PCIe slots on motherboards supporting simultaneous connections of multiple GPUs. Some GPU cards integrate multiple GPU

chips, such as ATI Radeon HD5970 and NVIDIA Tesla S1070. Therefore, scalability of optimized DGEMM on multiple GPUs is also important. Due to platform limitations, experiments run two MPI processes, each responsible for one GPU chip. Since key factors affecting DGEMM scalability are shared resource contention, such as PCIe bus and memory bus, using performance on two GPU chips to predict DGEMM scalability on multiple GPUs is feasible. The experiment attempts to predict DGEMM scalability on multiple GPUs through bandwidth contention between two GPU chips. The experiment profiles 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 on DP-2GPU relative to DP-1GPU bandwidth. First, we consider PCIe bandwidth contention between load2 and store1. Figure 9 shows average bandwidth reduction, with the y-axis representing normalized relative bandwidth. As shown, effective bandwidth for load2 and store1 reaches 89% and 56% of DP-1GPU, respectively. Since PCIe transfers are more frequent and require larger data sizes (matrix C is larger than the sum of matrices A and B), store1 bandwidth degradation is more severe. As mentioned in Section 3.3, to fully utilize the pipeline, the mult1 kernel further partitions sub-matrices into smaller sub-blocks, with each store1 operating on a smaller matrix sub-block. During mult1 execution, several 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 for DGEMM running on a single GPU chip, store1 already has high PCIe bandwidth occupancy. Therefore, when scaling to two GPU chips, PCIe bus contention becomes more severe. However, PCIe effective bandwidth degradation during load2 is not as severe as store1 because load2 pipelines with mult1 kernel across work units, unlike the intra-work-unit pipelining of store1 and mult1, making PCIe requests less frequent. Additionally, load2 transfers matrix sizes of $(m + n) \times k$, 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 also exists between two GPU chips because data transfers also occur between application space and CAL remote memory. Figure 10 shows effective memory bandwidth for load1 and store2 decreases by 8% and 14%, respectively. Similar to PCIe bandwidth degradation, store2 bandwidth reduction 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 shows that effective bandwidth reduction still causes 11% efficiency degradation in DP-2GPU. This indicates that shared resource contention (PCIe bandwidth and memory bandwidth) prevents some data transfer processes from overlapping with the mult1 kernel. As GPU count increases, bandwidth requests become more frequent, intensifying bandwidth contention' s impact on overall perfor-

mance. 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:** Due to PCIe bandwidth limitations, DGEMM scalability on multiple GPU cards on the same motherboard is constrained. In DGEMM, both load2 and store1 use PCIe bandwidth. As Figure 7 shows, these two operations account for 60% of DGEMM' s total data transfer time. Figure 8 experimental results show efficiency degradation on two GPU chips (DP-2GPU) due to contention. As GPU count increases, PCIe contention intensifies, further impacting DGEMM efficiency.
- **Conclusion 2:** Improving system memory bandwidth will enhance GPU-only DGEMM performance (DP-1GPU and DP-2GPU). Although both load1 and store2 occupy memory bandwidth, Figure 10 shows they are less sensitive to memory bandwidth contention. As Figure 7 shows, these components are not the major part of data transfer time. While pinned memory usage can avoid load1 and store2 contention in some cases, it requires that data not be reallocated and that data size is within pinned memory limits. Our work demonstrates that this transfer overhead can also be reduced through algorithmic optimization.

4.3.3 Scalability with 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. CPU computing capability cannot be ignored when optimizing compute-intensive programs like DGEMM. In hybrid DGEMM HB-2GPU implementation, matrices are first divided into two equal parts, each computed by a CPU/GPU pair. In each CPU/GPU pair, we adopt the partitioning algorithm from reference [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 the causes of efficiency degradation.

Figure 8 shows DGEMM efficiency on the hybrid CPUs/GPUs system. Figure 11 analyzes the performance contribution of the CPU component (denoted CPU-HB) to HB-2GPU. Pure CPU DGEMM (denoted Pure-CPU) serves as comparison, computing the same matrix scale as CPU-HB. The figure shows hybrid DGEMM CPU component performance is 22% lower than Pure-CPU. This is because GPU-computed DGEMM in hybrid DGEMM also needs to copy data from application space to CAL remote memory, interfering with CPU DGEMM computation and intensifying CPU memory contention. We conclude:

- **Conclusion 3:** Improving system memory bandwidth helps reduce memory contention, thereby enhancing 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 will help reduce

memory contention.

Another cause of efficiency reduction is load imbalance between CPU and GPU. We adopt a heuristic partitioning strategy to keep CPU and GPU execution time differences within a threshold. Based on multiple experiments, this paper uses 0.1 seconds as the threshold. Figure 12 plots relative differences between CPU and GPU execution times, with the y-axis representing $((GPU\ time - CPU\ time)/CPU\ time)$. This slight imbalance causes minor overall performance reduction in hybrid DGEMM (about 5%).

5 Related Work

Previous works have optimized DGEMM on GPUs for matrix sizes smaller than GPU memory. N. Nakasato proposed a new DGEMM kernel [15] based on HD5870, achieving 87% of GPU peak performance. Our optimized DGEMM kernel reaches 94% efficiency on one Cypress chip of HD5970. Chris Jang' s GATLAS auto-tuner [20] uses auto-tuning to enhance portability across different GPU architectures and intends to be called in real applications. GATLAS also only focuses on cases where matrices fit in GPU memory, so no direct way exists to call GATLAS in real applications yet. V. Volkov and J. Demmel implemented single-sided matrix factorizations (LU, QR, etc.) on hybrid CPU/GPU systems [12], distributing factorization processes to CPU and GPU for simultaneous execution. However, matrix multiplication within still only targets cases where matrices fit in GPU memory, with no CPU-GPU data transfer. Many other works optimize specific programs on GPU without considering data transfer, not enumerated 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 the Jacobi algorithm on hybrid CPU/GPU architecture [23]. They considered CPU-GPU data transfer, but hybrid program performance improvement is only 8%. DaQi Ren and Reiji Suda implemented large-scale matrix multiplication on multi-core CPU/GPU systems [25]. Their focus was energy consumption, using CPU multi-threading for optimization. When one thread waits for GPU memory access completion signals, CPU starts a new thread to execute other tasks. This approach enables GPU and CPU to compute simultaneously, but data transfer still stalls the entire execution, wasting CPU and GPU computing capability. Christian Feichtinger et al. implemented parallel Lattice Boltzmann methods on hybrid CPU/GPU clusters [24]. They minimized transfer data volume by only transmitting boundary values of partial differential equations. They considered load imbalance as one reason for limited performance improvement. Y. Ogata et al. proposed a model-based heterogeneous Fast Fourier Transform library for CPU/GPU systems [27]. This model better guides task partitioning between CPU and GPU. We adopt the adaptive partitioning algorithm from reference [21] to solve this problem 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 include data transfer in total time but also optimize it with pipeline algorithms, enabling this overhead to overlap with computation. Therefore, hybrid DGEMM can be called in real applications. Through our optimization, hybrid DGEMM achieves 844 GFLOP/s performance, corresponding to 80% efficiency. Canqun Yang et al. proposed using DGEMM computation to hide data transfer overhead [21], employing optimizations like data loading pipelining and data output pipelining. We further developed data placement optimization to improve DGEMM kernel performance and established a finer pipeline. This additional optimization strategy improves DGEMM performance by 74%, becoming the most important optimization. Additionally, we analyze shared resource contention (especially PCIe bus and system memory) and scalability of optimized DGEMM on 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), obtaining a new pipeline algorithm. In the pipeline, we overlap DGEMM kernel execution on GPU with data transfer processes. Optimized DGEMM achieves 408 GFLOP/s on an ATI HD5970 Cypress chip with 88% efficiency. On ATI HD5970, performance is 758 GFLOP/s with 82% efficiency. On heterogeneous CPU/ATI GPU systems, hybrid DGEMM reaches 80% of peak performance–844 GFLOP/s. Comparison with kernel performance shows optimized DGEMM efficiency on a single GPU chip approaches peak, with limited further optimization space. However, when DGEMM scales to multiple GPUs and CPUs, efficiency is impacted, primarily by shared resource contention, especially PCIe and system memory contention. Through experiments and analysis, we draw three conclusions: (1) Due to PCIe bandwidth limitations, DGEMM scalability on multiple GPU cards on the same motherboard is constrained; (2) Improving system memory bandwidth will enhance GPU-only DGEMM performance (DP-1GPU and DP-2GPU); (3) Improving system memory bandwidth helps reduce system memory contention, thereby enhancing 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, 2010, Institute of Computing Technology, lijiajia@ict.ac.cn
Xingjian Li: Master' s student, 2008, Institute of Computing Technology
Guangming Tan: Associate Professor, Institute of Computing Technology

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