Parallel Processing Letters
© World Scientific Publishing Company

ACHIEVING NATIVE GPU PERFORMANCE FOR OUT-OF-CARD LARGE DENSE MATRIX MULTIPLICATION

JING WU
*Department of Electrical and Computer Engineering
and Institute for Advanced Computer Studies, University of Maryland, College Park
College Park, Maryland 20742, USA*

and

JOSEPH JAJA
*Department of Electrical and Computer Engineering
and Institute for Advanced Computer Studies, University of Maryland, College Park
College Park, Maryland 20742, USA*

Received September 2014
Revised June 2015
Communicated by S. Rajasekaran

ABSTRACT

In this paper, we illustrate the possibility of developing strategies to carry out matrix computations on heterogeneous platforms which achieve native GPU performance on very large data sizes up to the capacity of the CPU memory. More specifically, we present a dense matrix multiplication strategy on a heterogeneous platform, specifically tailored for the case when the input is too large to fit on the device memory, which achieves near peak GPU performance. Our strategy involves the development of CUDA stream based software pipelines that effectively overlap PCIe data transfers with kernel executions. As a result, we are able to achieve over 1 and 2 TFLOPS performance on a single node using 1 and 2 GPUs respectively.

Keywords: Dense Matrix Multiplication; GPU; Heterogeneous Platforms

1. Introduction

Dense matrix operations are widely used as building blocks in many scientific and engineering computations. Double precision dense matrix multiplication (DGEMM), constituting the most important routine of the LINPACK benchmark used to rank the top 500 supercomputers, has been a major research focus for both academic researchers and processor vendors. Currently clusters consisting of nodes based on multicore CPU/many-core accelerators are very popular among the top 500 supercomputers due to their peak FLOPS performance and their energy efficiency. High performance native DGEMM libraries with high efficiency (up to 90%) are often provided by vendors of CPUs [1], GPUs [8, 5], and other accelerators such as

Xeon Phi coprocessor [2]. However when it comes to the out of card performance on a heterogeneous node, the great efficiency is typically compromised due to the substantial overhead caused by the memory transfers between the CPU and the GPU.

In this paper, we present a scalable scheme for accelerating DGEMM on heterogeneous CPU-GPU platforms, focusing on the case when the input is too large to fit on the device memory. Our scheme exploits hardware and software features of the CPU-GPU heterogeneous nodes and employ asynchronous CUDA stream based on software pipelines to achieve close to the best possible native CUDA BLAS DGEMM performance (CUDA BLAS assumes that both the input and output reside on the device memory).

The rest of the paper is organized as follows. Section II provides an overview of the hardware and software features that are heavily used in this work, followed by a brief introduction of the most popular DGEMM libraries and related literature. Section III starts by discussing popular blocking schemes which are essential to high performance DGEMM followed by a description of our blocking scheme. Section IV provides details about our software pipeline which enables near peak performance. Section V illustrates the performance of our strategy in terms of achievable FLOPS and scalability.

2. Overview

Our target systems are CPU-GPU heterogeneous platforms consisting of multi-socket multi-core CPU and one or more GPU accelerators. The input data is much larger than the size of the device memory and is assumed to be initially held in the CPU memory. At the end of the computation, the output data must reside in the CPU memory as well.

We use two testbeds for our work. The first is a dual socket quad-core Intel Xeon X5560 CPU with 24GB main memory and two NVIDIA Tesla C1060 cards each with 4GB device memory - we refer to this testbed as the "*Nehalem-Tesla* node", after the codename of the CPU and the architecture of the GPU respectively. The second is a dual socket octal-core Intel Xeon E5-2690 with 128GB main memory and two NVIDIA Tesla K20 cards each with 5GB device memory - we refer to this testbed as the "*Sandy-Kepler* node" (we use Sandy rather than Sandy Bridge for brevity). Data transfers between the CPU main memory and the GPU device memory are carried out by PCIe Gen2x16 bus: unidirectional for the *Nehalem-Tesla* node (compute capability 1.3) and bidirectional for the *Sandy-Kepler* node (compute capability 3.5).

2.1. *CUDA Programing Model*

The CUDA programming model assumes a system consisting of a host CPU and massively parallel GPUs acting as co-processors, each with its own separate memory [6]. The GPUs consist of a number of Streaming Multiprocessors(SMs), each

Table 1. GPUs specification & Compiler, Library configuration

Node	*Nehalem-Tesla*	*Sandy-Kepler*
CPU Name	Intel Xeon X5560	Intel Xeon E5-2690
Sockets x Cores	2x4	2x8
DRAM	24GB	128 GB
STREAM BW [4]	37GB/s	73 GB/s
icpc & MKL Lib	2013	2013
GPU Name	Tesla C1060	Tesla K20
Device Mem Size	4GB GDDR5	5GB GDDR5
Device Mem BW	102.4GB/s	208GB/s
SMs x SPs	30x8	13x192
PCIe bus	PCIe Gen2x16	PCIe Gen2x16
Bi-directional PCIe	No	Yes
PCIe achievable BW	5.4GB/s H2D 5.3GB/s D2H	5.7GB/s H2D 6.3GB/s D2H
CUDA driver	304.88	319.23
CUDA SDK	5.0	5.5
CUDA DGEMM Peak	75.3 GFLOPS	1053 GFLOPS

of which containing a number of Streaming Processors (SPs or cores). The GPU executes data parallel functions called kernels using thousands of threads. The mapping of threads onto the GPU cores are abstracted from the programmers through - 1) a hierarchy of thread groups, 2) shared memories, and 3) barrier synchronization. Such abstraction provides fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism and is based on similar hardware architecture among generations. Details of the CUDA programming model can be found at [6] and we will only refer to the aspects that are key to our optimization scheme. In this work, we are concerned with Tesla C1060 and K20 whose main features are summarized in Table 1. Note that, for the Tesla K20, the L1 cache and the shared memory per SM share a total amount of 64KB on-chip memory whose ratio is configurable as 1:3, 1:1 or 3:1.

2.2. *PCIe bus*

The CPU and the GPU communicate through the PCIe bus whose theoretical peak bandwidth is 8GB/s on PCIe Gen2x16 on both platforms. PCIe bus transfer typically uses pinned memory (explicitly) to get better bandwidth performance because the GPU cannot access data directly from the pageable host memory (A temporary pinned memory is implicitly used as a staging area otherwise.) The bandwidth difference between using a pinned memory versus pageable memory varies among platforms depending on whether both CPU and GPU support the same generation of the PCIe bus, their own DRAM bandwidth, etc.. For example, on one of our nodes, the hardware-to-device (H2D) bandwidth is around 3.3GB/s if we use pageable memory and similarly, the bandwidth of device-to-memory (D2H) transfer is around 3GB/s; on the other hand, using pinned memory we can reach 5.7GB/s for

H2D transfer and 6.3GB/s for D2H transfer. However, we should be careful not to over-allocate pinned memory so as not to reduce overall system performance. Another technique that is typically used is to combine many small transfers into a large transfer to eliminate most of the per-transfer overhead and latency . This is especially important when the GPU device memory can only hold a subset of the input dataset.

2.3. *Asynchronous Streams*

CUDA supports asynchronous concurrent execution between host and device through asynchronous function calls - control is returned to the host thread before the device has completed the requested task [6]. Data transfer and kernel execution from different CUDA streams can be overlapped when memory copies are performed between page-locked host memory and device memory. Some devices of compute capability of 2.x and higher (K20 in our evaluation) can perform memory copy from host memory to device memory (H2D) concurrently with a copy from device memory to host memory (D2H). With a careful orchestration of the CPU work and CUDA streams, we essentially establish a CPU-GPU work pipeline of depth five in which computation and communication are organized in such a way that each GPU accelerator (K20) is always busy executing a kernel that achieves 1TFLOPS performance. Since the data access pattern forces us to batch/pack small segments of data, we make use of the pinned memory to achieve better PCIe bus bandwidth especially since as we need to use such a staging area anyway.

2.4. *Existing CPU/GPU DGEMM Libraries*

Almost all vendors have developed optimized DGEMM libraries that exploit their processor architectures quite effectively. The list includes the DGEMM libraries developed by Intel [3] and AMD for their multicore CPUs, the NVIDIA CUBLAS_DGEMM for the NVIDIA GPUs, and Intel's DGEMM library optimized for the Xeon Phi coprocessor. None of these libraries address heterogeneous platforms and each seems to have been tailored for a particular architecture, even from generations to generations. In particular, the libraries in [8] and [7] are optimized DGEMM for earlier CUDA Tesla architecture GPUs and later Fermi architecture GPUs, respectively.

3. Overall Matrix Multiplication Blocking Scheme

3.1. *General Blocking Scheme*

Blocking is a common strategy for most optimized DGEMM implementations which involves decomposing the matrices into blocks of appropriate sizes and performing block-wise operations.

The general double-precision matrix multiplication is defined as $C = \alpha AB + \beta C$, where A, B and C are respectively $M \times K$, $K \times N$ and $M \times N$ matrices, and where α

and β are constants. Our DGEMM kernel assumes row-major format and the main strategy is to decompose the matrix multiplication into a set of outer-products defining jobs to be assigned to asynchronous CUDA streams. To define the jobs and the CUDA streams, two major issues have to be addressed. The first is how to alleviate the PCIe bus bandwidth limit for the CUDA streams and the second is how to maintain near peak performance for all the block matrix computations while overlapping data transfers over the PCIe bus with these computations.

The DGEMM kernel can be decomposed in a block form as follows:

$$C_{ij} = \alpha \sum_{k=0}^{K/bk} A_{ik}^{(0)} B_{kj}^{(0)} + \beta C_{ij}^{(0)},$$

where the superscript (0) indicates that the initial input data residing on the CPU, A_{ik}, B_{kj} and C_{ij} are sub-blocks of matrices A, B and C of sizes $bm \times bk$, $bk \times bn$ and $bm \times bn$ respectively. A computation of the C_{ij} defines a job that consists of s basic steps defined by:

$$C_{ij}^{(1)} = \alpha^{(0)} A_{i0}^{(0)} B_{0j}^{(0)} + \beta^{(0)} C_{ij}^{(0)}.$$

$$C_{ij}^{(k+1)} = \alpha^{(0)} A_{ik}^{(0)} B_{kj}^{(0)} + C_{ij}^{(k)}, \text{ where } k = 1, ..., s-1$$

where $s = K/bk$, $\alpha^{(0)} = \alpha$, and $\beta^{(0)} = \beta$. That is, for each step k of the job C_{ij}, we compute the matrix multiplication of $C_{ij}^{(k+1)} = \alpha^{(0)} A_{ik}^{(0)} B_{kj}^{(0)} + \beta C_{ij}^{(k)}$, for $k = 0, ..., s-1$, where $\beta = 1$ for $k \neq 0$ steps and β is equal to the original $\beta^{(0)}$ in the calling function when $k = 0$.

We use this decomposition to achieve the following:

(1) We select the block sizes that will allow us to make use of the fast native GPU DGEMM kernels, and to balance the execution time of such kernel with the transfer time of the blocks over the PCIe bus.
(2) The result of step k is the input for step $k+1$ - this reduces the pressure on the PCIe bus as sub-matrix C_{ij} is reused. We only need to load $C_{ij}^{(0)}$ before the first step and store $C_{ij}^{(s)}$ after the last step. Hence, the cost of moving block C_{ij} is amortized and the PCIe bus bandwidth requirement is alleviated.
(3) Since separate streams are responsible for computing separate submatrices C_{ij}, we avoid synchronization overhead between streams and make the decomposition strategy scalable - in fact, strongly scalable as we will show later.

3.2. *Lower Bounding Block Sizes*

In this section, we analyze the conditions on the dimensions bm, bk and bn of blocks A_{ik}, B_{kj} and C_{ij} so as to satisfy bus bandwidth requirement (within limit) for near peak GPU performance throughput.

Consider the computation of C_{ij} and how the corresponding three matrix blocks are transferred through the PCIe bus into the GPU memory. While this can be viewed in a similar vein as "caching" in a CPU, there are some major differences. First, the GPU has a much larger "cache" size (5GB for Tesla K20 vs 256KB per core for Xeon E5-2690). the second difference relates to a much smaller memory bandwidth (5.7GB/s H2D and 6.3GB/s D2H v.s. 73GB/s for dual-socket Xeon E5-2690 STREAM benchmark). The third, and perhaps the most important, a much higher experimentally peak DGEMM library *FLOPS* rate (1053 *GFLOPS* (CUBLAS 5.5) on Tesla K20 vs 320 *GFLOPS* on dual socket Xeon E5-2690). Based on these observations, we derive our bounds as follows. We note that the space needed to store the three matrix blocks is given by $8\,bytes \cdot (bm \cdot bn + bm \cdot bk + bn \cdot bk)$, and such data will be used to perform $2bm \times bk \times bn$ floating point operations. To keep the GPU execution units at full speed, assuming a peak performance of $GFLOPS_{peak}$ of CUBLAS_DGEMM (1.053 *TFLOPS*), the resulting PCIe bus host to device transfer bandwidth has to satisfy:

$$BW_{peak} \geq BW_{req} = \frac{8 \cdot (bm \cdot bk + bm \cdot bn + bk \cdot bn) \times 2^{-30}}{\frac{2 \cdot bm \cdot bk \cdot bn \times 10^{-9}}{GFLOPS_{peak}}}$$

To develop an intuition into the PCIe bus bandwidth requirement stated above, assume that $bm = bn = bk = dim$, which yields $BW_{req} = (12 \times 931.3/dim)\ GB/s$ which has to be $< 5.7GB/s$ (H2D). This inequality assumes that PCIe bus is not shared among GPUs which is the case in our testbed - each GPU is directly connected to a CPU via PCIe as we are using a dual-socket CPU. Otherwise, it may need to be adjusted according to the target platform by simply dividing the number of GPUs that are sharing a single PCIe bus. Solving this inequality, we get that $dim > 1960$ - which when rounded to $dim = 2000$, the required space for the three blocks is merely 0.09GB.

Next let's consider the more general case, that is, the block dimensions are distinct. This results in:

$$4 \times (\frac{1}{bm} + \frac{1}{bn} + \frac{1}{bk})\frac{2^{-30}}{10^{-9}} \times GFLOPS_{peak} < BW_{peak}$$

Substituting our *Sandy-Kepler* platform's $GFLOPS_{peak} = 1053$ and $BW_{peak} = 5.7$, we get

$$\frac{1}{bm} + \frac{1}{bn} + \frac{1}{bk} < \frac{1}{688}$$

This inequality provides an overall guideline for determining the block sizes - any block dimension smaller than 688 on our platform implies an under-utilization of the GPU kernels - how severe the under-utilization depends how bad the chosen block size is. Note that no matter what kind of blocking scheme and data reuse are employed, at least one block needs to be transferred from the host memory. This also indicates, if we would like to use CUDA GPUs to accelerate host-stored dense matrix multiplication, there is a minimum dimension requirement, for example, on Tesla K20, $min\{M, N, K\} > 688$ for achieving a good efficiency relative to the native CUBLAS/DGEMM.

In Figure 1 we evaluate the GFLOPS performance of the LAPACK_DGEMM on the CPU and the CUBLAS_DGEMM (CUDA 5.5) using one K20 on our platform. Given the peak CPU and GPU performances, the dense matrix multiplication procedures achieve more than 90% efficiency on both the CPU and the GPU for reasonably large data sizes. Substituting the evaluated GFLOPS for a given square matrix size into the PCIe bandwidth requirement formula, we get the actual PCIe bus bandwidth requirement corresponding to that size, which is also plotted in Figure 1.

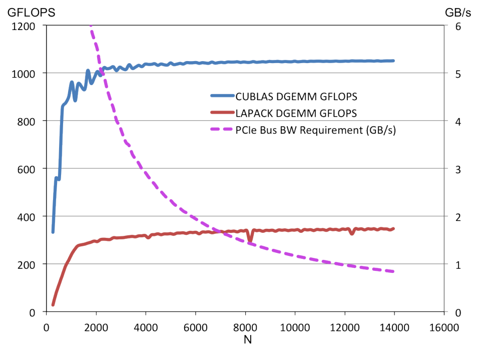

Fig. 1. Performance of DGEMM libraries and Corresponding PCIe BW requirement

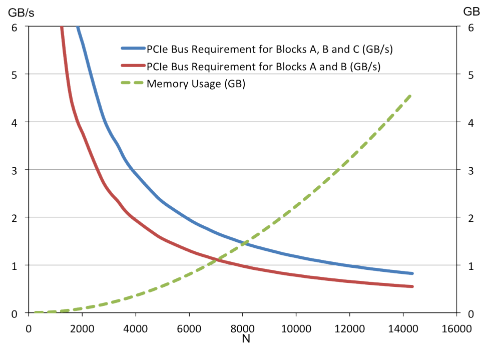

Fig. 2. PCIe BW Requirement of Staging Outer-Product

Since we are staging outer-product to compute the matrix block C_{ij}, the PCIe bus bandwidth requirement is highest for the first step as only blocks A_{ik} and B_{kj} need to be loaded in later steps. Depending on the number of steps, if the chosen block dimensions bm, bn and bk only satisfy the PCIe bus requirement for two blocks, this would result in an idle period of GPU in the pipeline. From Figure 2, we can see that, as the square matrix size increases, the memory requirement on the PCIe bus drops rapidly. At the same time, we need to note that as the matrix size increases, the required device memory space increases , which has several implications. The first is that a fewer number of concurrent jobs (computing individual C_{ij} blocks) can be scheduled as concurrent CUDA streams on the same GPU as different streams need separate space to store their matrix blocks. The second is that the GPU idle time before its first stream starts to execute increases, which should be optimized (minimized under constraints).

As a result, we follow the following rules for selecting the block dimensions bm, bn and bk:

(1) Using CUBLAS_DGEMM kernels to compute block matrix multiplication should achieve at least 1TFLOPS performance.
(2) The space requirements for the matrix blocks A_{ik}, B_{ik} and C_{ij} should be large enough as stated in the PCIe bus bandwidth requirement formula, but not be too large so that we are able to accommodate a number of concurrent CUDA streams to allow the overlapping of memory copy and kernel execution.

3.3. *Packing Data for PCIe Transfers*

Packing matrix blocks into micro- or macro- architecture friendly formats is another popular technique used in optimized DGEMM. In addition to the size of each matrix block, the data layout is another issue to consider. In [1], Goto et al. packed matrix blocks that can be fit into the L2 cache so as to achieve the minimal number of TLB entries. In [2], Heinecke et al. further extended the packing scheme to the so-called "Knights Corner-friendly" matrix format (column-major format for sub-block A and row-major format for sub-block B) for their Xeon Phi coprocessor. The "Knights Corner" strategy achieves 89.4% efficiency.

Similarly, we pack each of the matrix blocks needed for each step of the outer-product (C_{ij}) into row-major order form using multithreaded memory copy to the pinned memory. This strategy allows us to reach the peak experimental PCIe bus bandwidth. We note that, due to the memory capacity difference between the CPU main memory and the GPU device memory, we have to perform some synchronization to avoid data hazards in the pinned memory. We make use of the CPU main thread to accomplish such synchronization. Our multithreading in fact improves the CPU system memory to pinned host memory copy bandwidth, as the CPU packing step contributes to the overall runtime.

4. Multi-stage Multi-stream Software Pipeline

CUDA allows the use of streams for asynchronous memory copy and concurrent kernel executions to hide long PCIe bus latency [6]. A stream is a sequence of commands that execute in order; different streams may execute their commands out of order with respect to one another or concurrently. To optimize performance, we need to overlap the execution of the kernels and the PCIe bus transfers from different streams. We explicitly allocate a relatively small amount of pinned host memory and use multi-threading to move data between the large pageable host memory and the pinned host memory, which will enable us to achieve high bandwidth PCIe bus transfers.

We will start by describing a simple five stage task that computes a single matrix block multiplication, followed by a description on how to organize multiple CUDA streams into a multi-stage multi-stream software pipeline. The scheme will then be extended to the most general case while accommodating data reuse requirements that were mentioned earlier.

4.1. *A Simple Five-stage Task*

Consider the task of simply computing $C_{ij}^1 = A_{ik}^0 B_{kj}^0 + C_{ij}^0$, where matrix block sizes are $bm \times bn$, $bm \times bk$ and $bk \times bn$ respectively. One "*task*" here corresponds to one "*step*" for the job corresponding to the computation of C_{ij} as discussed in Section III.B. Such a task requires a single execution of a CUBLAS_DGEMM kernel call on the data blocks that have been brought from the CPU host memory to the device memory via the pinned host memory. Once the kernel terminates, the result C_{ij}^0 is moved back to the CPU host memory via the pinned host memory. Specifically, this task can be executed by a five-stage pipeline as follows:

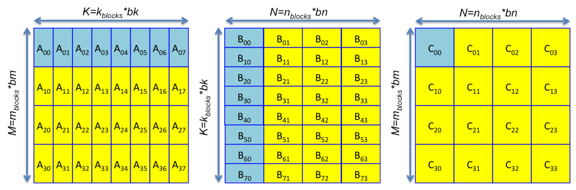

Fig. 3. Matrix Blocking Scheme

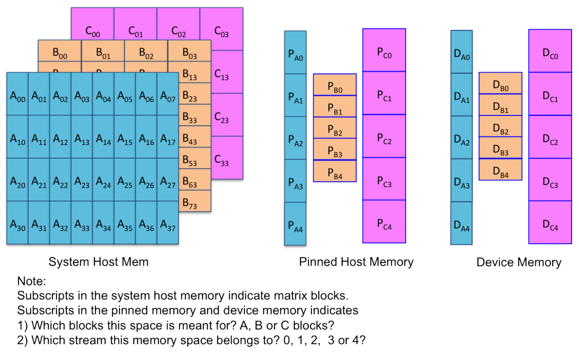
System Host Mem
Pinned Host Memory
Device Memory
Note:
Subscripts in the system host memory indicate matrix blocks.
Subscripts in the pinned memory and device memory indicates
1) Which blocks this space is meant for? A, B or C blocks?
2) Which stream this memory space belongs to? 0, 1, 2, 3 or 4?

Fig. 4. Memory Space Mapping (Assuming 5 Streams)

(1) Memory copy of blocks A^0_{ik}, B^0_{kj} and C^0_{ij} from the system host memory to the pinned host memory using multi-threading. We call this operation S2P memory copy.
(2) Asynchronous CUDA memory copy from the pinned host memory to the device memory for blocks A^0_{ik}, B^0_{kj} and C^0_{ij}. Such an operation is referred to as P2D memory copy.
(3) CUBLAS_DGEMM kernel execution to compute $C^1_{ij} = A^0_{ik}B^0_{kj} + C^0_{ij}$.
(4) Asynchronous CUDA memory copy from the device memory to the pinned host memory for block C^1_{ij}. This operation will be referred to as D2P memory copy.
(5) Memory copy of block C^1_{ij} from the pinned host memory to the system host memory, possibly using multi-threading. This operation will be referred to as P2S memory copy.

To execute a single five-stage task, we allocate pinned host memory and the device memory to hold blocks of C_{ij}, A_{ik} and B_{kj}. Assuming we can accommodate five independent tasks on the platform, the corresponding memory mapping is illustrated in Figure 4.

The time spent on each of the five stages can differ significantly depending on the block sizes, bus transfer bandwidth, and kernel performance, an issue that will be addressed next.

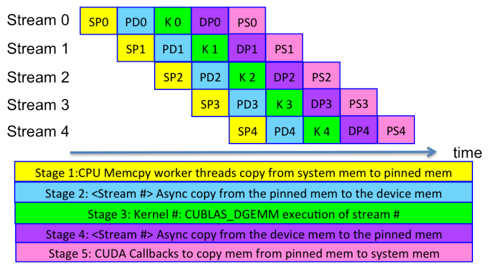
Stream 0
Stream 1
Stream 2
Stream 3
Stream 4
SP0 PD0 K 0 DP0 PS0
SP1 PD1 K 1 DP1 PS1
SP2 PD2 K 2 DP2 PS2
SP3 PD3 K 3 DP3 PS3
SP4 PD4 K 4 DP4 PS4
time
Stage 1:CPU Memcpy worker threads copy from system mem to pinned mem
Stage 2: <Stream #> Async copy from the pinned mem to the device mem
Stage 3: Kernel #: CUBLAS_DGEMM execution of stream #
Stage 4: <Stream #> Async copy from the device mem to the pinned mem
Stage 5: CUDA Callbacks to copy mem from pinned mem to system mem

Fig. 5. Basic 5-stage Pipeline

Given the 5-stage pipeline just described, we aim at executing 5 streams concurrently as illustrated in Figure 5. To achieve this, the execution times of the different stages of the pipeline should be somehow balanced. The kernel execution stage (stage 3) is expected to be the most time consuming since the throughput of the pipeline is expected to match the native GPU DGEMM performance. This time-consuming step will be matched with appropriate choices of block sizes bm, bn, and bk as described earlier by the PCIe bandwidth lower bound formula. Hence we expect the execution time of stages 2 and 4 to match the kernel execution time. The execution times of stages 2 and 4 are expected to be faster, but that should be fine as long as there are no idle time gaps between the kernel executions among multiple streams as we will show later.

4.2. *Multiple Streams of Multi-Stage Pipeline in the General Case*

A straightforward approach to handle dense matrix multiplication would be to assign tasks-based jobs of C_{ij} to streams based on the decomposition formula in a given order.

$$C_{ij} = \alpha \sum_{k=0}^{K/bk} A_{ik}^{(0)} B_{kj}^{(0)} + \beta C_{ij}^{(0)}$$

Assume that we have already decided on appropriate block sizes as illustrated in Figure 3. bm, bn and bk for a problem size of M, N and K. We note that $m_{blocks} = M/bm$, $n_{blocks} = N/bn$ and $k_{blocks} = K/bk$. We can define job_{id} as the computation of C_{ij} using the index mapping $job_{id} = i \times n_{blocks} + j$. Note in particular that our index mapping attempts to minimize the TLB misses as the large input data need to be accessed from the CPU main memory. Assume that a number (SN) of CUDA streams with $stream_{id} = 0, ..., (SN\text{-}1)$ are executed concurrently. We assign the $m_{blocks} \times n_{blocks}$ jobs to the SN streams in a round-robin manner, modulo SN. For every assigned job, the stream is responsible for moving, computing and storing the final result of C_{ij} into the host memory. The computation of C_{ij} involves a sequence of (k_{blocks}) of DGEMM function calls, that is, k_{blocks} basic tasks. We will describe later how what type of synchronization we need so that we can schedule consecutive jobs to the same stream.

Figure 5 shows a simplified example of a 5-stage pipeline consisting of 5 CUDA asynchronous streams. In this example, each stream is handling a single job that includes a single task. Each stream uses its own pinned memory space and device memory space to store the A_{ik}, B_{kj} and C_{ij} blocks. ($k_{blocks} = 1$ in this figure.)

We use the matrix blocking scheme in Figure 3 to explain the resulting streams. for the general multiple-task-per-job case ($k_{blocks} > 1$) According to our job_{id} and $stream_{id}$ relationship, we assign the computation of C_{00} to $stream_0$, which consists of 8 (k_{blocks}) iterations (sequence) of the basic five-stage stream tasks from $k = 0, ..., 7$.

As shown in Listing 1, the movement of block C_{ij} is controlled by conditional statements that ensure data reuse. In general, at least one of the three blocks of A_{ik}, or B_{kj}, or C_{ij} may be reused.

Listing 1. Tasks Per Job

```
for (int k = 0; k < 8; k++)
{
  // stage 1
  if(k==0) Memcpy_S2P_C(0,0);
  Memcpy_S2P_A(0,k);
  Memcpy_S2P_B(k,0);
  // stage 2
  if(k==0) Memcpy_P2D_C(0,0);
  Memcpy_P2D_A(0,k);
  Memcpy_P2D_B(k,0);
  // stage 3
  CUBLAS_DGEMM
  // stage 4
  if(k==7) Memcpy_D2P_C(0,0);
  // stage 5
  if(k==7)  Memcpy_P2S_C(0,0);
}
```

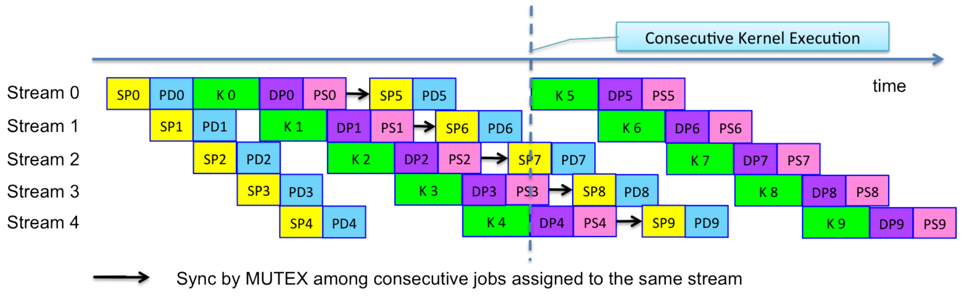

Fig. 6. CPU-GPU Software Pipeline

The main goal in our design of the software pipeline is to ensure the continuous full utilization of the GPU near its peak performance. A key is to maintain a steady supply of data blocks to each GPU on our platform. Note that CUDA asynchronous streams can execute out of order with respect to each other but function calls within the same stream have to execute in order. For matrix multiplication problem, we orchestrate the streams and their function calls in such a way that CUBLAS_DGEMM calls are executed immediately one after the other, each resulting in near-peak performance per GPU. The overall scheduling of the multi-stage multi-stream pipeline is described in Listing 2. Note that we are able to achieve full utilization of the GPU for a much larger problem size than the device memory capacity using a memory mapping illustrated in Figure 4.

Listing 2. Multi-Stage Multi-Stream Pipeline

```
int jobs = m_blocks*n_blocks;
for (int i = 0; i < jobs; i +=SN)
{ // tid = task_id;  sid = stream_id;
  for(int tid=0; tid<k_blocks; tid++){
    for(int sid=0; sid<SN; sid++) {
      job_id = i+sid;
      if(job_id>=jobs) break;
      Wait_for_CPU_S2P(job_id, tid);           // stage 1
      Launch_AsyncMemcpy_P2D(job_id, tid);     // stage 2
      CUBLAS_DGEMM(job_id, tid);               // stage 3
      Launch_AsyncMemcpy_D2P(job_id, tid);     // stage 4
      if(job_id+SN>=jobs)                      // stage 5
      Update_last_flag(sid);
      Launch_CUDA_Callbacks_P2S(job_id, tid);
    }
  }
}
```

4.2.1. *Synchronization*

Memory reuse requires that mechanisms are put in place to avoid data hazards. We use MUTEX to achieve this goal. As we allocate different memory spaces for different streams, a data hazard can only happen within the same stream. We first assign flags for each stream in each potential block that can be over-written (A, B and C respectively). These are each protected by a MUTEX after which we combine those flags appropriately to minimize the overhead of synchronization. Note that such synchronization overhead is typically "invisible' as long as it does not impede the CUBLAS_DGEMM executions as we have enough active CUDA streams to hide the synchronizations within the same stream's tasks/jobs as the black arrows illustrate in Figure 6. Specifically, we insert CUDA stream callbacks, executed as a CPU thread after previous CUDA kernel calls associated with that stream are completed. In the callbacks, we set the status flag to be "0" notifying the CPU worker threads to resume their memory copy work from the system host memory to the pinned host memory, which would flip the status to "1" and wait for the execution of another callback. A simplified pipeline with one task per job is illustrated in Figure 6.

4.3. *Multi-stage Multi-stream Pipeline For Small K*

So far we have focused on matrix multiplication with relatively similar dimension $< M, N, K >$ values, which gave us a significant number of choices for block sizes.

In this section, we tune our scheme for the case when M and N are much larger than K. That is, matrix A is skinny, matrix B is fat, and matrix C is large and almost square. This case is frequently used in parallel dense matrix operations such as LU, QR and Cholesky factorizations. The challenge to the strategy described earlier is two-fold: 1) there is much less flexibility in selecting the block size for the dimension K; and 2) the large size of the input and output matrix C puts much more pressure on the bi-directional PCIe bus bandwidth than the almost square case.

As discussed before, in order to achieve near peak performance, no blocking dimension should be smaller than 688 for platforms using PCIe Gen2x16 bus. Hence, we assume $K > 688$ and we use $K = 1024$ as an example. Due to the inequality bound, we necessarily have $k_{blocks} = 1$, which yields this simple outer product $C_{ij} = A_{i0}B_{0j}$. This means that for each CUBLAS_DGEMM kernel execution, we would have to load and store a C_{ij} block, which is unavoidable. Due to the fact that K, aka bk is small, we are left with no choice but to have larger bm and bn to guarantee that the inequality will still hold. As a result, this gives us a really big C_{ij} block to transfer in both directions, in addition to the relatively smaller size A and B blocks.

We optimize such a situation in two ways. First, we assign jobs to the streams for which blocks of A or B could be reused in different jobs. For example, we assign the computation of C_{0j} to stream 0 and keep matrix A_{00} in the device memory throughout the computation of C_{0j} other than swap it out. Second, we use two components of the pinned host memory space for matrix C: one as Write-Combining Memory to conduct the H2D memory transfers for better bandwidth utilization; and the other one as default cacheable memory for the other way around as write-combining memory for D2H memory transfers.

5. Performance

In this section, we evaluate the performance of our proposed multi-stage pipeline based approach on two different platforms. Detailed specifications of the platforms are listed in Table 1.

5.1. *Square Matrix Multiplication*

The overall performance of our general blocking scheme for a range of matrix sizes ranging from N=1K to 52K on the *Sandy-Kepler* and *Nahalem-Tesla* nodes is shown in Figures 7 and 8 respectively. We compare the GFLOPS performance of our implementations using 1 and 2 GPUs to the Intel MKL multi-threading DGEMM using all the CPU cores available on the *Sandy-Kepler* node.

Similar to previous work, the number of FLOPS is determined by the expression $2 \cdot MNK$, where A is of size $M \times K$, B of size $K \times N$, and C of size $M \times N$. On the *Sandy-Kepler*, our approach greatly exploits the optimized performance of the CUDA DGEMM library and achieves 1 or 2 TFLOPS for all reasonably large data

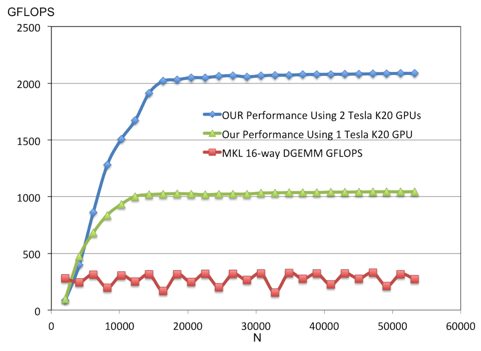

Fig. 7. DGEMM Perf. on *Sandy-Kepler*

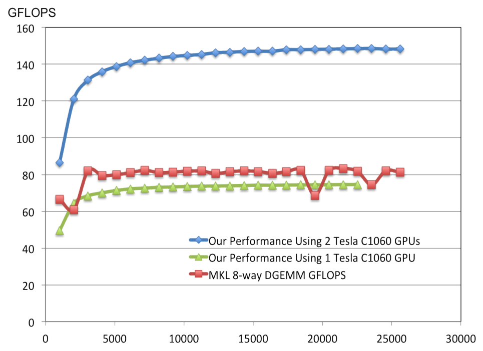

Fig. 8. DGEMM Perf. on *Nehalem-Tesla*

sizes by using either one or two GPUs. Such a performance is substantially better than the corresponding performance on the multi-core CPUs. In addition, unlike the native CUDA DGEMM library, whose problem size is limited by the device memory capacity, our approach essentially gives an illusion of a device memory size equal to the CPU host memory while delivering the same CUBLAS_DGEMM GFLOPS performance. In order to illustrate the generality of our scheme, we evaluate the same implementation on the *Nahalem-Tesla* node. Due to its weak double precision performance - a peak native library performance of 75.3GFLOPS - we are able to nearly match the native performance and double it on two GPUs.

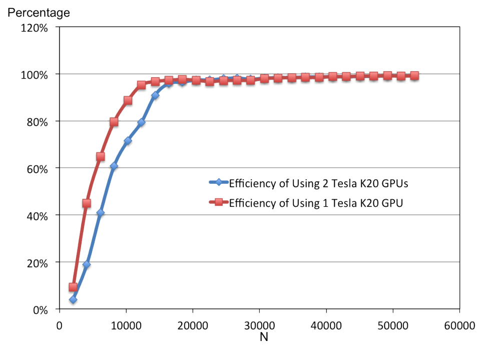

Fig. 9. Efficiency on the *Sandy-Kepler* Node

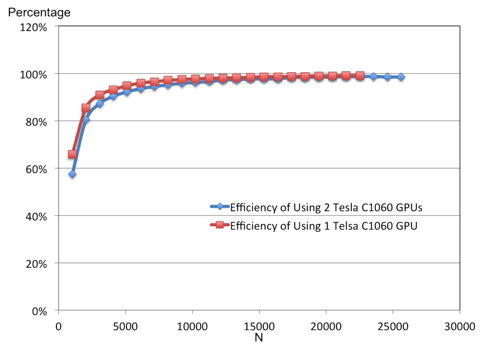

Fig. 10. Efficiency on the *Nehalem-Tesla* Node

To shed more light on the effectiveness of our multi-stream software pipeline, we define the efficiency as follows:

$$efficiency = \frac{GFLOPS_{achieved}}{GFLOPS_{peak\ lib\ perf}}$$

We demonstrate the efficiency of our scheme in Figures 9 and 10. As we can see from both figures, when the problem size is reasonably large, our software pipeline is quite efficient and brings almost all of the native CUDA DGEMM library performance out to the host memory. The same type of efficiency is obtained for both nodes in spite of their differences.

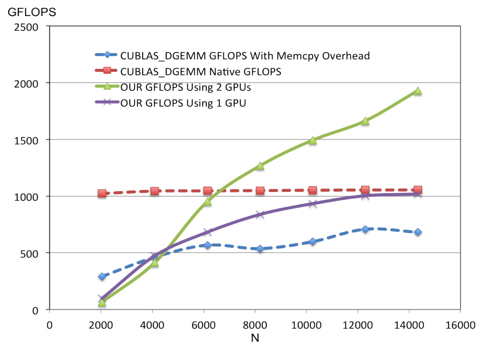

Fig. 11. Smaller Size Perf. on *Sandy-Kepler*

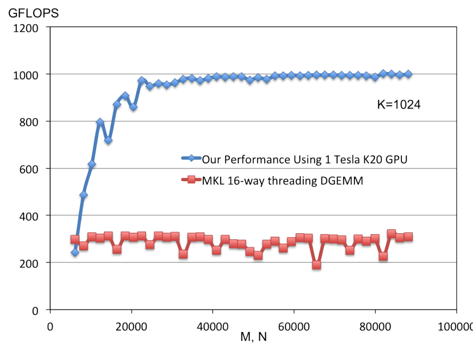

Fig. 12. Small K DGEMM Perf. *Sandy-Kepler*

We note that the decomposition used is not always beneficial for small data size, which was anticipated by our inequality bound. We demonstrate the performance of relatively smaller size matrices in Figure 11. Though the native CUBLAS_DGEMM performance on K20 is more than 1TFLOPS for all problem size of $N > 2K$, transferring the input from the CPU host memory and the output back to the CPU contribute a significant overhead. In fact, when the the problem size is fairly small, say $N = 2K$, we may simply want to use a straightforward CUDA DGEMM call. Notice that in this case the problem fits on the device memory, while the focus of this paper is on problems that cannot fit on the device memory.

5.2. *The Case of Dense Matrix Multiplication for Skinny A and Fat B*

We now illustrate the performance for the case when matrix A is skinny and matrix B is fat. We fix $K = 1024$ and vary $M = N$ value over a wide range. Our strategy works extremely well and shows scalability similar to the square case. The results are shown in Figure 12.

Similarly, we demonstrate the GFLOPS performance and the efficiency as shown in Figure 13.

6. Conclusion

We have developed a pipelining strategy to carry out dense matrix multiplication for the case when the input size is much larger than the size of the device memory. Our strategy achieves almost the same native CUDA DGEMM library performance over a wide range of large sizes. We achieve more than 1 teraflops on a single GPU and twice the performance on two GPUs, thereby illustrating the possibility of using the GPUs with a memory size equal to the size of the main memory on the host machine. The key to this performance is the careful selection of the block sizes and the orchestration of the various stages of a CUDA multi-stream that ensures continuous GPU executions near peak performance. Our results raise the possibility

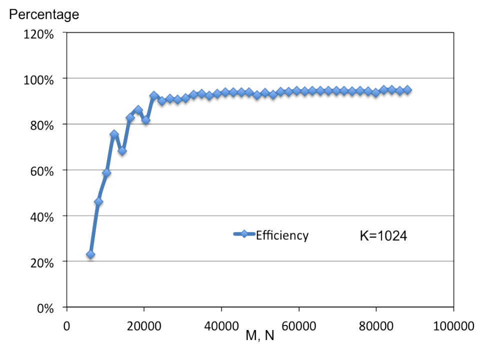

Fig. 13. Small K DGEMM Efficiency on *Sandy-Kepler* Node

of carrying out various dense matrix operations on very large matrices stored in the CPU memory while achieving native GPU performance on matrices that fit on the device memory.

Acknowledgment

This work was partially supported by an NSF PetaApps award, grant number OCI-0904920, the NVIDIA Research Excellence Center award to the University of Maryland, an NSF Research Infrastructure Award, grant number CNS-0403313, and by the NSF National Socio-Environmental Synthesis Center.

References

[1] K. Goto and R. A. v. d. Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.*, 34(3):12:1–12:25, May 2008.
[2] A. Heinecke, K. Vaidyanathan, M. Smelyanskiy, A. Kobotov, R. Dubtsov, G. Henry, A. G. Shet, G. Chrysos, and P. Dubey. Design and Implementation of the Linpack Benchmark for Single and Multi-node Systems Based on Intel Xeon Phi Coprocessor. *Parallel and Distributed Processing Symposium, International*, 0:126–137, 2013.
[3] Intel. Math Kernel Library. `http://developer.intel.com/software/products/mkl/`.
[4] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http://www.cs.virginia.edu/stream/.
[5] NVIDIA Corporation. *CUBLAS Library User Guide*. NVIDIA, v5.5 edition, 2013.
[6] NVIDIA Corporation. NVIDIA CUDA C programming guide, 2013.
[7] G. Tan, L. Li, S. Triechle, E. Phillips, Y. Bao, and N. Sun. Fast implementation of DGEMM on Fermi GPU. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 35:1–35:11, New York, NY, USA, 2011. ACM.
[8] V. Volkov and J. W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.