2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing

A flexible and portable large-scale DGEMM library for Linpack on next-generation multi-GPU systems

David Rohr* and Volker Lindenstruth*

*Frankfurt Institute for Advanced Studies
Ruth-Moufang-Str. 1, Frankfurt am Main, Germany
Email: rohr@compeng.uni-frankfurt.de

***Abstract*—In recent years, high performance computing has benefitted greatly from special accelerator cards such as GPUs. Matrix multiplication performed by the BLAS function DGEMM is one of the prime examples where such accelerators excel. DGEMM is the computational hotspot of many tasks, among them the Linpack benchmark. Current GPUs achieve more than $1\,\mathrm{TFlop/s}$ real performance in this task. Being connected via PCI Express, one can easily install multiple GPUs in a single compute node. This enables the construction of multi-$\mathrm{TFlop/s}$ systems out of off-the-shelf components. At such high performance, it is often complicated to feed the GPUs with sufficient data to run at full performance. In this paper we first analyze the scalability of our DGEMM implementation for multiple fast GPUs. Then we suggest a new scheme optimized for this situation and we present an implementation.**

I. Introduction

The **Linpack** benchmark is a standard benchmark for measuring compute capabilities of supercomputers and it provides the basis for the Top500 list of the 500 fastest supercomputers in the world. Its standard implementation is called **HPL** (**H**igh **P**erformance **L**inpack) and many vendors provide a custom, modified version of HPL optimized for their hardware. **DGEMM** is a function provided by **BLAS** (**B**asic **L**inear **A**lgebra **S**ubprograms) libraries, which perfoms a matrix-matrix multiplication. Matrix-matrix multiplication is very compute intense, and DGEMM is the computational hotspot of many applications, among them the Linpack benchmark. Therefore, DGEMM has always been subject to sophistic optimizations. Most vendors deliver optimized DGEMM libraries for their hardware such as the Intel MKL for Xeon CPUs and the Xeon Phi, AMD ACML for Opteron CPUs, clAmdBlas for AMD GPUs, or cuBLAS for NVIDIA GPUs. Modern HPC servers are equipped with up to $256\,\mathrm{GB}$ of memory or even more. GPUs have much less but faster memory: usually $16\,\mathrm{GB}$ or less. Hence, it is often impossible to hold the whole problem in GPU memory. This paper is about DGEMM on GPUs for large matrices which fit in system memory but not in GPU memory.

The focus of DGEMM optimizations has traditionally been the DGEMM kernel, which performs the actual matrix multiplication. DGEMM for large matrices is usually processed through a splitting of the matrices in tiles which are processed in a pipeline one after another [2], [5]. We have shown that performance is independent of the tile size, if it exceeds a certain minimum [2]. These tiles must be transferred to the GPU and back to the host. Hence, in heterogeneous systems with hardware accelerators connected via PCIe, the DMA transfer and the system memory bandwidth are also important aspects. For single-GPU systems we have shown [2] that one can hide the transfer behind computation on the GPU. Looking ahead to next-generation systems, however, it is possible that compute capabilities of multi-GPU systems will exceed the DMA and the memory bandwidth. In this paper we describe a new way to organize the DGEMM operation, which saves bandwidth and CPU resources. We demonstrate that, in case of Linpack, our approach is scalable to at least four next-generation accelerators with $2\,\mathrm{TFlop/s}$ double-precision performance each, for an aggregate DGEMM performance of about $8\,\mathrm{TFlop/s}$ per compute node.

II. Related Work

Most DGEMM implementations for GPUs, those offered by hardware vendors as well as open source libraries and those presented in most papers, have their focus on the DGEMM kernel. Processing of large matrices that do not fit in GPU memory must either be handled by the user. Or the libraries use tiling approaches, which split the large matrix in small matrix tiles which are processed one after another. Although the large matrix is stored in a contiguous memory segment, its smaller submatrices are generally not stored in such a contiguous segment. Traditionally, many APIs did not offer a transfer for non-contiguous memory, or they provided only a slow solution for this. The standard solution is to pack data into contiguous temporary buffers prior to transfer.

Volkov et al. [11] was one of the first papers to discus DGEMM on NVIDIA GPUs thoroughly. With respect to DMA, the authors only show measurements for transfers of contiguous data and do not discuss the issue of matrices that do not fit in GPU memory. Nakasato [8] presents a fast DGEMM kernel on AMD Cypress GPUs but does not cover the case of huge matrices in host memory either. Our previous work in this field [2], [9] used a tiling approach with packing/unpacking, which was sufficient for today's clusters [1], [10]. However, we will show that this does not scale arbitrarily. Kurzak et al. [7] use a different approach. They have a queue in which they insert small DGEMM tasks fitting in GPU memory. However, they do not discuss in detail how they arrange the transfer. The Intel HPL [5] uses matrix tiling and packing/unpacking, in the

1066-6192/15 $31.00 © 2015 IEEE
DOI 10.1109/PDP.2015.89

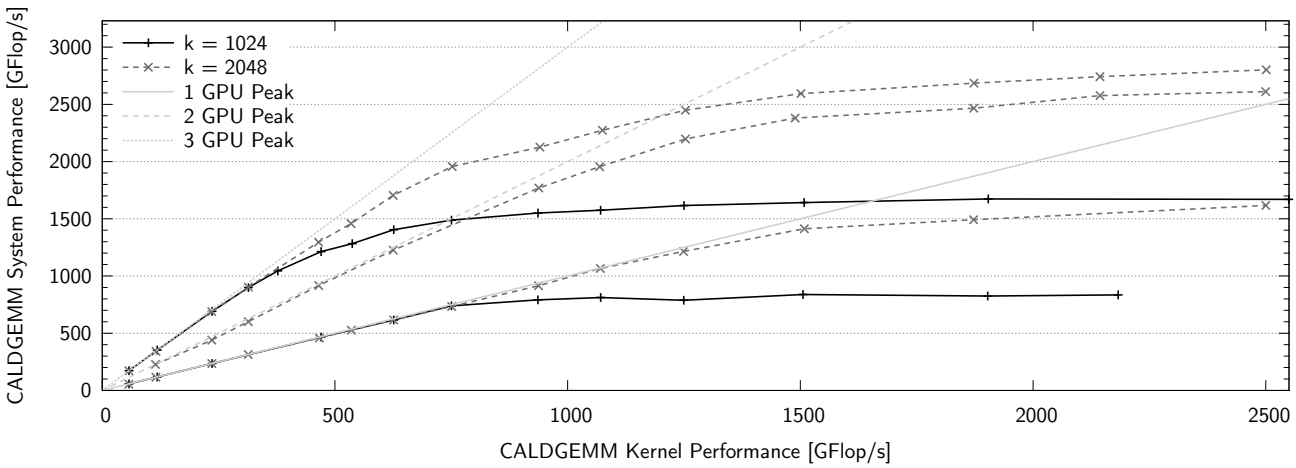

Fig. 1. Scalability of the initial CALDGEMM System Performance in relation to the CALDGEMM kernel performance. The plot shows the theoretical peak system performance with n GPUs as straight, light grey lines. (This is calculated as the kernel performance of one GPU times n.) The solid black lines and the dashed grey lines show the measured performances with a different number of GPUs (1 GPU for the lowermost line, 3 GPUs for the uppermost line) and for a different value of the parameter k (solid black for $k = 1024$, dashed grey for $k = 2048$).

same way we did up until now. For the Roadrunner cluster, IBM placed the entire matrix in device memory, avoiding any transfer [6]. Endo et al. [3] used tiling with non-rectangular matrices. This allows contiguous DMA transfers without packing/unpacking. However, it only works if the difference between host and device memory is not too large (due to matrix shape restrictions in DGEMM). So this approach does not scale to the memory sizes of $256\,\mathrm{GB}$ and beyond, which is our main focus here. Fatica [4] has shown that one can avoid the packing/unpacking of tiles by using GPU-registered host memory. Unfortunately, at the time that paper was published, the CUDA API employed was not able to register more than $4\,\mathrm{GB}$ of memory. Hence, for larger matrices packing/unpacking was still necessary. Fatica uses the CUBLAS helper functions as a black box to move data between CPU and GPU. Accordingly, the paper does not discuss the ramifications of non-contiguous data transfers. This was in fact formerly not needed since GPU kernels were considerably slower such that transfer performance was not that dominant. With no alternative at that time, that implementation is restricted to CUDA. We will present a more portable solution supporting CUDA as well as OpenCL.

None of these papers have their primary focus in PCI Express bandwidth. If it is discussed, only the DMA bandwidth itself is analyzed. In contrast, we will show that the main bottleneck lies in limited host memory bandwidth. We will suggest a new DMA scheme primarily to reduce load on host memory and on CPU, and not in order to improve DMA throughput.

III. Contribution of This Paper

We have already presented a GPU-enabled implementation of the Linpack benchmark called **HPL-GPU** which uses our GPU DGEMM library called **CALDGEMM** [2], [9]. In the following we present measurements of the scalability of our previous implementation and present an improved DMA scheme for multi-GPU systems.

DGEMM computes $\mathbf{C}_{\text{new}} = \alpha\mathbf{AB} + \beta\mathbf{C}$ with α and β scalars, $\mathbf{A}$ an $m \times k$ matrix, $\mathbf{B}$ a $k \times n$ matrix, $\mathbf{C}$ an $m \times n$ matrix, and where $\mathbf{C}_{\text{new}}$ is overwritten onto $\mathbf{C}$. If DGEMM is used within HPL-GPU, k is equal to the blocking size N_b of the Linpack algorithm and it is usually in between of 1024 and 2048. The values of m and n in Linpack are significantly larger (above 100000). By splitting the DGEMM loop over k in multiple calls, our algorithm works well for larger k, too. The computationally expensive part is the matrix-multiplication $\mathbf{AB}$, which the GPU should process. Values of $\mathbf{C}$ are only used in a single addition, so there is no sense in transferring $\mathbf{C}$ to the GPU for a single operation. It was thus a design decision of the original CALDGEMM implementation to calculate only $\mathbf{X} = \alpha\mathbf{AB}$ on the GPU, transfer $\mathbf{X}$ to the host, and then calculate $\mathbf{C}_{\text{new}} = \mathbf{X} + \beta\mathbf{C}$ on the CPU. This approach worked very well within HPL-GPU on the LOEWE-CSC [1] and SANAM [10] clusters and e. g. the Intel HPL [5] works the same way. On the one hand, this avoids the transfer of the original $\mathbf{C}$ matrix to the GPU. On the other hand, this results in four host-memory accesses per matrix element: the entry of $\mathbf{X}$ is stored in host memory by the GPU's DMA engine, it is then read by the CPU, the CPU in addition reads the corresponding entry of $\mathbf{C}$ to perform the addition, and finally the CPU stores the result. We have shown that in multi-GPU systems this can exceed the host memory bandwidth [9]. This can be mitigated by using a larger Linpack blocking size N_b (and thus a larger value of k) which reduces the required bandwidth. However, this works only to a certain degree, since a larger N_b increases the complexity of the CPU-based factorization of HPL-GPU. We consider 2048 – 2560 to be the upper limit for N_b. As a summary, it is important to find ways to reduce the system memory load.

We use Fig. 1 to analyze how well our initial approach scales to multiple fast GPUs of the next generation. In order to simulate a GPU with an arbitrary DGEMM performance, we use a DGEMM mockup: a fake kernel that simulates a certain performance. In detail, we have changed the inner loops in our standard DGEMM kernel such that it breaks the computation earlier simulating a faster kernel, or to run some calculations multiple times simulating a slower kernel. Naturally, the result

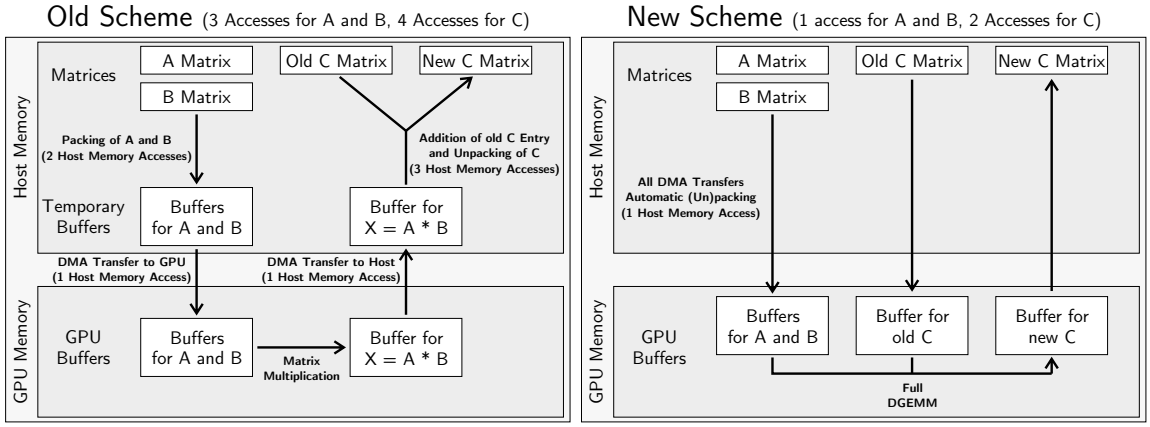

Fig. 2. Illustration of the data transport schemes with the old and the new approach. The old approach requires packing and unpacking of data to consecutive memory segments in special host buffers prior to the transfer. The new approach performs the transfer directlyand thus reduces the number of host memory transactions. It performs an automatic, implicit packing/unpacking of the data. Hence, it eliminates the CPU load caused by packing/unpacking. As a consequence, the new approach must move the addition of the old $\mathbf{C}$ matrix entry onto the GPU. Before, this was handled together with the unpacking of the data. (The scalars α and β are omitted for simplicity here. Host memory accesses are counted per matrix element.)

is no longer the correct matrix multiplication result, but we can simulate any performance. We use the actual DGEMM kernel as a basis in order to have a GPU memory access scheme similar to a real DGEMM. If we run with g GPUs with a simulated performance of p, the theoretical peak aggregate performance in the system is $g \cdot p$. This is indicated by the straight, light grey lines in Fig. 1 for one to three GPUs. The figure contains measurements with $k = 1024$ and $k = 2048$. (The line with two GPUs and $k = 1024$ is skipped for better readability.) It is visible that in any case the measurement curves (starting with very slow kernels from the left) first follow the straight lines indicating the peak performance. In this region, bandwidth is not a bottleneck. Then at certain points, the measurement curves become constant. This indicates the kernel performance for which bandwidth becomes the bottleneck. PCIe bandwidth is always below $4\,\mathrm{GB/s}$ and hence no limiting factor, system memory bandwidth is the problem. It is obvious that, due to the reduced bandwidth requirement mentioned before, the curves with $k = 2048$ scale better than those with smaller k. The value $k = 1024$ is infeasible already for two GPUs, while $k = 2048$ works well for two but not for three or more GPUs. All curves saturate before $1\,\mathrm{TFlop/s}$ kernel performance, hence this approach is not scalable to multiple fast next-generation GPUs.

IV. A new approach

In the following we assume the setting of DGEMM within HPL-GPU with about $256\,\mathrm{GB}$ of memory, which means $k \approx 2048$, $m, n \gg k$. This means the matrix $\mathbf{C}$ is much larger than $\mathbf{A}$ and $\mathbf{B}$, and with respect to the DMA transfer, $\mathbf{A}$ and $\mathbf{B}$ are negligible. To cope with multiple fast GPUs in the future, we suggest a scalable approach that performs the entire DGEMM calculation on the GPU. For this purpose, the original $\mathbf{C}$ matrix must be transferred to the graphics card. As already explained, the old approach requires four host memory transactions per matrix element. The new approach requires only two host memory transactions per entry: one for sending the initial $\mathbf{C}$ matrix-entry per DMA and one for receiving the new entry for $\mathbf{C}_{\text{new}}$. However, with the new approach $\mathbf{C}$ must be transferred to the GPU, which was not necessary before. Hence, this raises the required host-to-GPU bandwidth, which was negligible before as $\mathbf{A}$ and $\mathbf{B}$ are small, to roughly the same level as the GPU-to-host bandwidth, which is unchanged. Overall, this approach reduces the system memory load, which constitutes the bottleneck, by 50%. In contrast, it doubles the required aggregate PCI Express bandwidth. Since PCI Express works in full duplex mode and the maximum unidirectional bandwidth does not change significantly, this should not pose a problem.

This new approach also reduces the number of host memory accesses required for transferring the $\mathbf{A}$ and $\mathbf{B}$ matrices. Heretofore, three memory transactions were necessary (two for packing data, one for the actual transfer). Now, only one transaction for the transfer remains. However, since in Linpack $\mathbf{A}$ and $\mathbf{B}$ are much smaller than $\mathbf{C}$, the savings are much smaller than for $\mathbf{C}$. Figure 2 illustrates DMA transfer schemes and savings.

With the new approach, the actual matrix in host memory is both read and written directly by DMA. Hence, compared to the old version, two aspects become obsolete: first, the host side DMA buffers. Second, the pre- and postprocessing on the host, which packs and unpacks the DGEMM data and performs the addition of the $\mathbf{C}$ matrix. This approach also simplifies the host-scheduling significantly. In contrast, it poses high demands on the DMA capabilities of the GPU. First, the DMA transfer is not restricted to a small dedicated DMA buffer on the host but the DMA engine must access the entire host memory. Second, instead of a consecutive memory segment which contains the packed tile of the $\mathbf{C}$ matrix, the DMA transfer must work on submatrices of $\mathbf{C}$, where each line of the submatrix is a separate memory segment. While linear DMA memory copy normaly achieves close to peak bandwidth, this is not automatically true for submatrix transfers. Finally, the new approach needs full duplex operation. Both OpenCL and CUDA offer a DMA API that is capable of autonomically transferring submatrices from host memory to continuous GPU memory and vice versa. Such a transfer is called a **strided** transfer – in

contrast to a **linear** transfer of a consecutive memory segment. DMA transfers can be performed by the GPU DMA engine or by either a GPU kernel or a CPU thread via Zero-Copy. This offers the following methods for DMA transfers:

I Both transfer to GPU and back to host are performed by the GPU DMA engines prior to and after kernel execution.

II The GPU kernel directly reads and stores the C-entries from and to host memory via Zero-Copy. The DMA engines are not used.[1] (A and B are always stored in GPU memory since they require much more bandwidth. Naturally, this method effects the DGEMM kernel performance. Depending on the hardware, we have seen positive and negative effects.)

III The processor copies the matrix to the GPU via Zero-Copy. The transfer back is done via the DMA engine.[2]

We have found that we need multiple CPU cores writing to GPU memory to achieve enough PCIe bandwidth with Method III. Therefore, we did not investigate this method further. Method I stores the $\mathbf{C}$ matrix in global GPU memory via a PCI Express transfer. The kernel then reads it from there. This causes global GPU memory load equal to twice the PCI Express throughput. Method II skips this step. Hence, in theory, Method II is sovereign.

TABLE I. GPU DMA THROUGHPUT: THE TABLE SHOWS THE MEASURED PEAK DMA THROUGHPUT OF SEVERAL GPUS AND THE MAXIMUM ACHIEVABLE DGEMM PERFORMANCE.

GPU	Method	To GPU GB/s	To Host GB/s	Bidirectional GB/s	Max. Flops GFlop/s
GTX580	I	6.0	6.6	6.2	1598
GTX580	II			7.5	1932
M2070	I	5.9	6.6	8.7	2232
M2070	II			7.5	1932
6970	I	6.3	6.7	6.5	1659
6970	II			8.9	2287
S10000	I	8.8	12.2	14.7	3759
W9100	I	11.4	13.2	18.6	4775
W9100	II			10.7	2740

Table I lists uni- and bidirectional DMA throughput measurements employing Methods I and II. We calculate the DGEMM performance under the assumption the data are processed as fast as they are transferred to the GPU. This is the peak DGEMM performance achievable by reason of the available PCIe bandwidth. We have added these numbers to the table as Max. Flops. (These calculations assume $k = 2048$.) These measurements lead to the following considerations:

- The DMA engines (Method I) of the M2070, the S10000, and the W9100 can do full duplex transfers, the engines of the 6970 and GTX580 cannot.
- If full duplex DMA is supported, Method I can be faster. If not, Method II is faster (always using full-duplex).
- During the measurements, we saw certain incompatibilities, where some GPUs in combination with some drivers and some mainboards have shown significantly reduced DMA transfer speed for certain matrix sizes. Often, this affected only either the one or the other method.
- The new cards with PCIe 3.0 (S10000, W9100) show superior performance compared to the PCIe 2.0 cards.

In order to ensure the greatest flexibility for upcoming GPUs and driver improvements, we have implemented both Methods I and II for two APIs each: CUDA and OpenCL. We wanted to avoid code duplication and to keep the code portable and flexible. Hence, we have hidden all API specific components in derived classes of an abstract base class, which contains all the control logic. This approach will allow the adoption of new hardware or new APIs in the future. The new DMA scheme is optimized for fast multi-GPU systems. It is likely that for slower single-GPU systems the old approach will remain faster also in the future. Hence, we support both options.

Our original implementation uses preprocessing for transposing input matrices $\mathbf{A}$ and $\mathbf{B}$ and providing the input data in an optimal format for the kernel. The new implementations relocate these preprocessing tasks to conversion kernels on the GPU. After tiles of $\mathbf{A}$ and $\mathbf{B}$ have been transferred to the GPU, these conversion kernels reformat the input data for the DGEMM kernel. Our implementation caches the formatted tiles on the GPU [2], such that transfer and preprocessing of a tile is only performed once. In the same way as for the DMA transfer, since usually the input matrices $\mathbf{A}$ and $\mathbf{B}$ are relatively small compared to $\mathbf{C}$, the performance impact is negligible. Since no CPU-side pre- or postprocessing is needed, the implementation became much simpler. All DMA transfers and DGEMM kernels are queued into the CUDA or OpenCL command queues automatizing the scheduling. Only one check remains, which ensures that the input tile is already available in the GPU buffer before a kernel starts. (The caching of the tiles of $\mathbf{A}$ and $\mathbf{B}$ cannot be automatized by the command queue.)

V. BENCHMARKS

To analyze the scalability of the new scheme, the same technique as before with the mockup DGEMM is employed: a fake kernel simulates a particular kernel performance. For good performance Method I requires the memory to be allocated by the GPU runtime (instead of the malloc function). Method II enforces this restriction at all times. This means a large amount of memory must be registered for the GPU. We have tested the AMD OpenCL and the NVIDIA CUDA Framework on a Sandy Bridge EP based node of the SANAM cluster [10]. Fig. 3 compares the scalability of the CUDA version of the new DMA scheme with the old version. The OpenCL version shows almost identical results and is hence not included to keep the plot simle. Unfortunately, we had neither a test system with four professional grade cards nor a system with four PCI Express 3.0 cards available. The benchmarks are performed with four GTX580 GPUs running with PCI Express 2.0. Without a full-duplex DMA engine, Method I is limited to half duplex. Our measurement with the new DMA scheme shows a performance of up to roughly $2\,\mathrm{TFlop/s}$ per GPU. Method I saturates a bit earlier due to the half-duplex limitation. A comparison with Table I reveals that the single-GPU performance with Method I saturates at about $90\,\%$ of the limit posed by PCI Express bandwidth. Method II reaches $95\,\%$ even with four GPUs. Hence, our framework uses the available PCI Express bandwidth to the full extent.

[1] Naturally, this could be combined with Method I, e. g. by sending via I and receiving via II. However, measurements show that either of the two methods is faster for both sending and receiving and thus should be used exclusively.

[2] The processor can write to the GPU with high bandwidth using DMA write combining. Reading from the GPU is significantly slower, thus this method does not work well the other way around. Naturally, it could be used in combination with Method II instead of I, but if the kernel DMA is faster than the DMA engine, it can be used exclusively in any case.

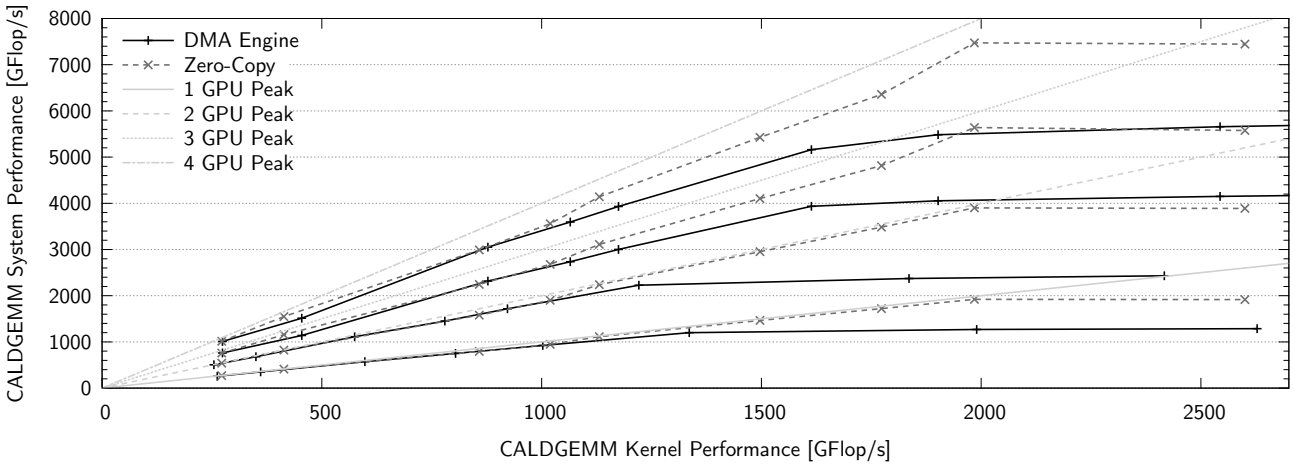

Fig. 3. Scalability of CALDGEMM with the new DMA scheme. The structure of the plot is similar to Fig. 1. It shows results for one to four GPUs. The solid black line shows the result using the DMA engine for the transfer while the dashed grey lines shows the result using Zero-Copy for the transfer. Curves for one (lowermost lines) to four (uppermost lines) GPUs are shown.

A comparison of Figures 1 and 3 reveals that with up to two GPUs, the old scheme can compete with the new version employing Method I. However, the old scheme is not capable of running more than two high performance GPUs of the next generation. With the Zero-Copy method, or with more than two GPUs, the new scheme easily outperforms the original implementation. It must be noted that even in situations where the performances of the old and the new schemes are even, the new scheme causes only half the memory load and utilizes less CPU cores leaving significantly more resources for concurrent tasks on the CPU. Finally, the new scheme scales to a kernel performance of nearly $2\,\mathrm{TFlop/s}$ for up to four GPUs, reaching $7.27\,\mathrm{GB/s}$ PCI Express bandwidth per GPU, $29.1\,\mathrm{GB/s}$ memory bandwidth on the host, and about the threefold total performance of the old implementation.

VI. Conclusion and Future Work

We have presented a new DMA transfer scheme for GPU-enabled large-scale DGEMM and we have implemented it for OpenCL and CUDA. We demonstrated that our CUDA version scales up to $2\,\mathrm{TFlop/s}$ per GPU in a four-GPU system. This aggregates to almost $8\,\mathrm{TFlop/s}$ of single-node DGEMM performance. The only limit for the performance in that case is the PCI Express bandwidth. Our DGEMM utilizes $95\,\%$ of the maximum PCIe bandwidth that we achieve with raw bandwidth measurements. This shows that on the DMA side, there is little margin for improvements. Overall, our new DMA scheme is ready for multiple fast GPUs of the next generation. For the future, we plan to investigate how transfer Method I compares to Method II on professional grade cards with full duplex DMA engines. PCI Express 3.0 could theoretically double the available bandwidth. It will be very interesting to observe how far the scheme can scale with this future hardware. Finally, AMD just released the professional grade cards of the Hawaii family, which peak at a double precision performance of $2.6\,\mathrm{TFlop/s}$. As soon as we have a fast DGEMM kernel for this card, we plan to reproduce the $2\,\mathrm{TFlop/s}$ per GPU measurement with a real kernel instead of the mockup fake kernel.

References

[1] M. Bach, J. De Cuveland, H. Ebermann, D. Eschweiler, M. Kretz, M. Pollok, D. Rohr, H. J. Lüdde, and V. Lindenstruth, "The LOEWE-CSC: A Comprehensive Approach for a Power Efficient General Purpose Supercomputer," in *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, 2013, pp. 1–17.

[2] M. Bach, M. Kretz, V. Lindenstruth, and D. Rohr, "Optimized HPL for AMD GPU and Multi-Core CPU Usage," *Computer Science - Research and Development*, vol. 26, no. 3-4, pp. 153–164, 2011.

[3] T. Endo, S. Matsuoka, A. Nukada, and N. Maruyama, "Linpack evaluation on a supercomputer with heterogeneous accelerators," *2010 IEEE International Symposium on Parallel Distributed Processing IPDPS*, pp. 1–8, 2010.

[4] M. Fatica, "Accelerating linpack with CUDA on heterogenous clusters," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, 2009, pp. 46–51.

[5] 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," *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 126–137, May 2013.

[6] M. Kistler, J. Gunnels, D. Brokenshire, and B. Benton, "Petascale computing with accelerators," in *Proceedings of ACM Symposium on Principles and Practice of Paralle Computing*, 2009, pp. 241–250.

[7] J. Kurzak, P. Luszczek, M. Faverge, and J. Dongarra, "LU Factorization with Partial Pivoting for a Multicore System with Accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 1, pp. 1–1, 2013.

[8] 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.

[9] D. Rohr, M. Bach, M. Kretz, and V. Lindenstruth, "Multi-GPU DGEMM and HPL on Highly Energy Efficient Clusters," *IEEE Micro, Special Issue, CPU, GPU, and Hybrid Computing*, 2011.

[10] D. Rohr, S. Kalcher, M. Bach, A. Alaqeeli, H. Alzaid, D. Eschweiler, V. Lindenstruth, A. Sakhar, A. Alharthi, A. Almubarak, I. Alqwaiz, and R. Bin Suliman, "An Energy-Efficient Multi-GPU Supercomputer," in *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications, HPCC 2014, Paris, France. IEEE*, 2014.

[11] V. Volkov and J. Demmel, "Benchmarking GPUs to Tune Dense Linear Algebra," in *SC 08 ACM/IEEE conference on Supercomputing Proceedings*, 2008, pp. 1–11.