Towards Distributed Heterogenous High-Performance Computing with ViennaCL

Josef Weinbub[1], Karl Rupp[1,2], and Siegfried Selberherr[1]

[1] Institute for Microelectronics, TU Wien, Vienna, Austria
[2] Institute for Analysis and Scientific Computing, TU Wien, Vienna, Austria

Abstract. One of the major drawbacks of computing with graphics adapters is the limited available memory for relevant problem sizes. To overcome this limitation for the ViennaCL library, we investigate a partitioning approach for one of the standard benchmark problems in High-Performance Computing (HPC), namely the dense matrix-matrix product. We apply this partitioning approach to problems exceeding the available memory on graphics adapters. Moreover, we investigate the applicability on distributed memory systems by facilitating the Message Passing Interface (MPI). Our approach is presented in detail and benchmark results are given.

1 Introduction

In the last couple of years, graphics adapters have been increasingly used for computations in the field of scientific computing, especially for linear algebra problems [1,2,6], but also to distribute data on several computing nodes powered by a graphics adapter [3]. The first major push forward has been accomplished by NVIDIAs CUDA library [14]. However, the CUDA library solely relies on NVIDIA products and therefore excludes other computing resources, like, graphics adapters from ATI/AMD and CPUs in general. OpenCL, on the other hand, overcomes this drawback as it offers a unified parallel programming model for a multitude of targets. The general concept of utilizing graphics adapters as processing units is referred to as General-Purpose computation on Graphics Processing Units (GPGPU).

To utilize OpenCL for linear algebra, ViennaCL [16] has been developed. The aim of this library is to provide a convenient means to access the vast computing resources of GPUs and multi-core CPUs.

Aside from the high-performance capabilities of graphics adapters, the limited memory of such computing targets is a drawback which hinders problems of considerable size to be computed on the graphics adapters. To further increase the execution performance, a distribution approach is required which facilitates the computational capabilities of several computing nodes.

We base our investigations on the dense matrix-matrix product within the ViennaCL library. We introduce an approach to overcome the memory restrictions on graphics adapters. Moreover, we distribute the workload on several computing nodes by MPI.

I. Lirkov, S. Margenov, and J. Waśniewski (Eds.): LSSC 2011, LNCS 7116, pp. 359–367, 2012.

The structure of this paper is as follows: Section 2 provides a short overview of the ViennaCL library. Section 3 discusses the memory constraints on graphics adapters. Section 4 rigorously discusses our approach to overcome the memory restrictions on a single graphics adapter. Moreover, a distributed approach based on MPI is presented. In Section 5 results are presented and discussed in detail.

2 ViennaCL

The Vienna Computing Library (ViennaCL) provides standard data types for linear algebra operations on GPUs and multi-core CPUs [4,5]. The library is based on OpenCL [12], from which ViennaCL inherits the unified parallel programming approach for personal computers, servers, handhelds, and embedded devices. The ViennaCL API is compatible to the Boost uBlas [9] library, which is a generic template class library with BLAS level 1, 2, 3 support. Therefore, existing uBlas implementations can be conveniently adapted to utilize the vast computing resources of, for example, GPUs. By now, additional support is provided for Eigen [10] and the Matrix Template Library 4 [15]. ViennaCL aims to provide a convenience layer for developing GPU accelerated applications.

3 Challenges of GPGPU

Although OpenCL and thus ViennaCL provide access to various computing targets, we focus our investigations on graphics adapters for two reasons. First, graphics adapters provide a massively parallelized environment, which can be facilitated for tasks in the field of linear algebra. Second, the memory constraints of graphics adapters introduce the need for special treatment. Let us consider these restrictions based on the dense matrix-matrix product (Equation 1).

$$C = A \times B; \quad A \in \mathbb{R}^{n \times m}, B \in \mathbb{R}^{m \times p}, C \in \mathbb{R}^{n \times p} \tag{1}$$

For this product the memory requirements (Equation 2) and the number of operations are as follows (Equation 3).

$$\texttt{memory:}(n \cdot m + m \cdot p + n \cdot p) \cdot \texttt{sizeof}(\mathit{double}) \tag{2}$$

$$\texttt{operations:}(2 \cdot n \cdot m \cdot p) \tag{3}$$

If we consider $n = m = p =: N$, it can clearly be seen that the memory complexity is $\mathcal{O}(N^2)$ and the complexity in regard to the required operations is $\mathcal{O}(N^3)$. For example, $N = 10\,000$ would yield $2.4 \cdot 10^9$ Bytes (~ 2.3 GBytes) of memory and $2 \cdot 10^{12}$ floating point operations.

On the contrary, the matrix-vector product offers the same complexity in the computation as in the memory requirements, which would be $\mathcal{O}(N^2)$ for $n = m = p =: N$. Consequently, this makes the matrix-matrix product much more interesting for investigations in regard to computational performance.

The memory requirements hinder the immediate application of large scale computations on graphics adapters, as the available memory of mere consumer level graphics adapters is typically much smaller, for example, the NVIDIA Geforce GTX 570 (1.2 GB) or the AMD Radeon HD 6970 (2 GB).

To tackle this fact, high-end workstation solutions are available which offer a considerable larger memory, like the NVIDIA Tesla C2050 (3 GB) or the Tesla C2070 (6 GB). These products naturally help to ease the problem, but still, they cannot even remotely compete with the system memory of workstations, let alone computing clusters. These computing environments typically offer system memory in the range from 32 GB to 128 GB per node. Although the memory restrictions limit the immediate application of large scale problems, the computing power of graphics adapters is due to the massive parallelized architecture of interest to the scientific community. Graphics adapters offer a high-performance per value in contrast to expensive cluster systems. Consequently, this introduces the need to overcome the memory restrictions to access the vast computing resources of graphics adapters which, however, requires special treatment of algorithms, for example, a matrix decomposition approach for the dense matrix-matrix product.

4 Our Approach

This section presents our approach which is based on two steps. First, we introduce a decomposition for the dense matrix-matrix product to overcome the memory restrictions of graphics adapters for a single computing node. This approach enables computing problem sizes with ViennaCL, which would normally be impossible due to the memory limitations. Second, a distribution approach based on MPI is presented, which enables to facilitate heterogenous computing nodes.

4.1 Overcoming the Memory Restrictions

To be able to process matrix products of large matrix sizes on memory restricted graphics adapters, a partitioning approach is used. These partitioned parts are then sequentially fed to the ViennaCL matrix product algorithms. Figure 1 depicts this approach.

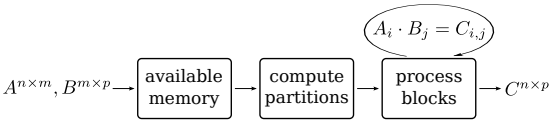


Fig. 1. Overview of the partitioning approach. The problem is partitioned into blocks based on the input matrix sizes and the available memory. Each block is processed by the ViennaCL matrix product algorithm and therefore contributes a subset to the overall solution matrix.

First, the available memory of the OpenCL device, in our case the graphics adapter, is computed. Then, the required partitions of the input matrices A and B are computed according to the available memory and the problem size. Blocks are extracted from the input matrices A_i, B_j and processed sequentially, based on the partitioning information. The computed partial results $C_{i,j}$ are collected and finally returned as overall result C. In the following the three different modules depicted in Figure 1 are discussed in some detail.

To be able to partition an input problem for a specific computing target, the available memory must be known. Unfortunately, OpenCL does not provide functionality which directly allows to determine the available memory of the device. OpenCL provides a means to extract two different memory values of the device by the `clGetDeviceInfo()` function, namely the global memory and the maximum allocable memory accessed by the `CL_DEVICE_GLOBAL_MEM_SIZE` and the `CL_DEVICE_MAX_MEM_ALLOC_SIZE` flag, respectively. However, both values cannot be interpreted as the allocable memory available on the device. The global memory refers to the total amount of memory physically available on the graphics adapter. This value cannot be allocated as the global memory exceeds the OpenCL internal limitations for maximum memory allocations and the host system allocates some of the graphics adapters memory too. On the contrary, the provided maximum allocable memory value yields an upper bound for a single allocation, while several allocations easily exceed this value. In other words, the OpenCL-provided maximum allocable memory value does not actually represent the maximum allocable memory.

Therefore, we use a simple algorithm which evaluates a reasonable allocable memory size specifically for the matrix-matrix product. A fraction of the maximum allocable memory is used as the initial value of this algorithm which is iteratively incremented until the memory can indeed not be allocated anymore. The last allocable memory value is than used as the available memory value of the OpenCL device. Figure 2 depicts the principle of our algorithm.

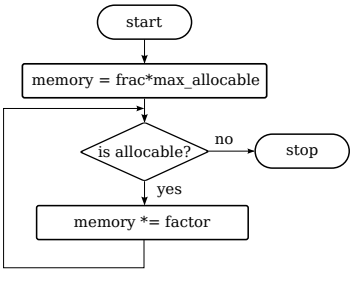


Fig. 2. Flowchart of our memory algorithm. The algorithm starts from a low value and iteratively increases the memory by a certain factor. The value which is last known to be allocable is returned.

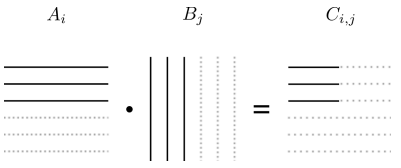


Fig. 3. The matrices A, B, C are partitioned to fit on the graphics adapters memory

The problem can be partitioned in such a way that it fits into the memory of the graphics adapter, based on the computed available memory estimate.
To maximize the performance of our ViennaCL implementation the problem is partitioned in regard to maximum memory utilization, as the execution performance increases with the problem size.

Aside from the memory constraints, the partitioning algorithm has to obviously respect the mathematical rules of matrix multiplication. Therefore, the rows of the matrix A and the columns of the matrix B are partitioned as depicted in Figure 3.

4.2 Distributed Computing

When considering larger problems, the workload for the matrix products can be distributed on available nodes. Each node is governed by a ViennaCL implementation capable of dealing with matrix products of considerable size, as introduced in Section 4.1. The distribution of the matrices is realized using Boost MPI. The matrix A for both matrix products is distributed evenly between the nodes by partitioning the rows. For our numerical experiments, the matrix B is distributed as a whole to all nodes. Note that in the case that B does not fit on the node, additional partitioning has to be applied in accordance to the memory availability on the respective computing nodes. Figure 4 depicts the MPI distribution approach in which, without loss of generality, we assume that the total number of entries in A is larger or equal to that of B.

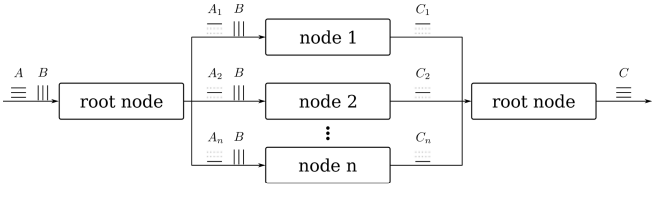


Fig. 4. Overview of the matrix distribution for different computation nodes. The matrix B is transferred as a whole to all nodes, whereas the matrix A is row-wise partitioned and evenly distributed.

Table 1. Specifications of the small-scale MPI computing environment. The CPU properties C and T denote the number of cores and threads, respectively.

MPI node	CPU	system memory	GPU
root	INTEL i7 960 4C/8T	12 GB	-
1	AMD Phenom II X4 965 4C/4T	8 GB	NVIDIA Tesla C2050
2	AMD Phenom II X4 955 4C/4T	8 GB	NVIDIA GTX 470
3	INTEL Core2 Q9550 4C/4T	4 GB	NVIDIA GTX 580

For the matrix storage we use a linear memory approach based on the standard C++ Standard Template Library (STL) vectortype, for which tuned Boost MPI transmission implementations are available.

The matrices A and B are stored in row-major and column-major order, respectively, since these memory layouts both favor the MPI based transmission efficiency and the memory layout of the OpenCL devices.

5 Results

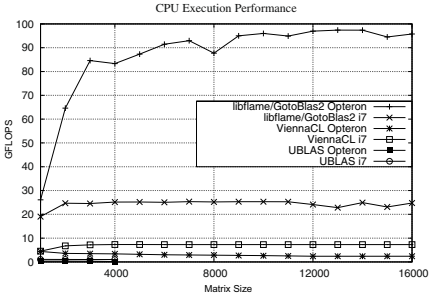
This section discusses the gathered results based on our approach, for two different computing environments. The first environment is composed of consumer level computing targets as depicted in Table 1. All systems are driven by a Funtoo Linux 64-bit distribution and NVIDIA drivers of version 260.19.36.

The second environment consists of one node of our professional MPI environment. The node is powered by four AMD Opteron 8435 CPUs, each providing six cores yielding a total number of 24 cores, and 128 GB of system memory. Our benchmarks are based on the double precision floating point datatype.

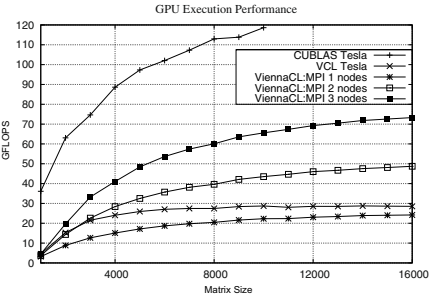
It is important to note that due to the OpenCL back-end, the same implementation is used to run on the CPU and the GPU driven systems. To be able to use CPUs as OpenCL computing targets, the AMD Accelerated Parallel Processing Standard Development Kit (APP SDK) 2.3 is used [8].

Figure 5a depicts the results for two CPU driven systems, namely the root node as depicted in Table 1 and the AMD Opteron cluster node. The standalone ViennaCL implementation is investigated without MPI distribution. We compare our CPU based results with reference implementations based on Boost uBLAS [9] and on libflame [13,7] driven by the GotoBLAS2 [11] library which is specifically tuned for the individual targets. Our results show that our implementation executed on the INTEL i7 target (8 GFLOPS[1]), although significantly slower than the reference libflame/GotoBLAS2 implementation (25 GFLOPS), scales adequately in reference to the uBLAS implementation (1 GFLOPS). Note that although the GotoBLAS2 library has been compiled to support 8 threads, it only utilized 4 threads on the INTEL i7 CPU, omitting the 4 additional threads provided by the Hyperthreading technology which is due to the missing support

[1] Giga Floating Point Operations per Second.



(a) Comparison of the execution performance on CPU-based targets. Our approach scales appropriately relative to the uBLAS implementation on the INTEL i7 target. However, as the quad-socket AMD Opteron node does not scale like the INTEL i7 target, it can be concluded that this quad-socket system is not fully supported by the OpenCL back-end.



(b) Comparison of the execution performance on GPU-based targets. The scaling of our MPI distribution approach as well as the MPI overhead can be identified. The reference CUBLAS implementation outperforms our standalone approach. However, it is due to the memory requirements restricted to matrix sizes of approximately 10000.

Fig. 5. Comparison of the execution performance on CPU- and GPU-based targets

for this specific technology by the GotoBLAS2 library. The quad-socket AMD Opteron cluster node is not fully utilized by the OpenCL back-end provided by the APP SDK. The uBLAS implementation yields 0.4 GFLOPS which is, compared to the INTEL i7 CPU, largely due to the significantly lower clock rate.

The ViennaCL implementation, however, reaches a mere 3 GFLOPS which, considering the higher performance achieved by the INTEL i7 target, indicates that the quad-socket AMD Opteron system is not efficiently utilized, because when restricted to six threads approximately the same performance can be achieved.

Apparently, such multi-processor computing targets are not fully supported by the APP SDK. Although our MPI distributed ViennaCL computation is capable of being executed on our cluster, it is skipped for benchmarking investigations due to the current state of support as depicted.

Figure 5b presents benchmark results based on GPU targets. We compare our approach on a single ViennaCL node with the MPI distribution approach as depicted in Table 1 and with a reference implementation based on CUBLAS [14]. The CUBLAS and our standalone ViennaCL implementations are executed on the root node (Table 1). The MPI distribution overhead can be identified when comparing MPI node 1 (crosses) with the standalone implementation (stars). The scaling behavior of our distribution approach can be recognized when analyzing the execution performance of one to up to three nodes. Although the reference CUBLAS implementation performs significantly faster than our standalone ViennaCL implementation, it is restricted to matrices of approximately

the size of 10000 due to the required memory. On the contrary both our standalone variant as well as our MPI variants, are capable of processing matrices of significantly larger problem sizes.

6 Conclusion

We discussed the challenges of computing large dense matrix-matrix products on graphics adapters. We introduced an approach to overcome this restriction based on the ViennaCL library. Moreover, we presented a distribution approach based on MPI to facilitate the power of several OpenCL computing nodes. We provided benchmark results for CPU and GPU driven systems, each compared to reference implementations. The scaling as well as the communication overhead of our MPI based distribution approach has been presented and discussed. A limited support of the APP SDK for quad-socket AMD Opteron systems has been identified. In regard to the CPU based investigations our results depict a reasonable scaling behavior relative to the reference single-core uBLAS implementation. Our GPU powered MPI distribution scales well for one to up to three nodes.

Acknowledgments. This work has been supported by the European Research Council through the grant #247056 MOSILSPIN. Karl Rupp gratefully acknowledges support by the Graduate School PDETech at the TU Wien. The authors thank NVIDIA for providing a Tesla C2050.

References

1. Agullo, E., et al.: Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. Journal of Physics: Conference Series 180 (2009)
2. Bell, N., Garland, M.: Efficient Sparse Matrix-Vector Multiplication on CUDA. Tech. Rep. NVR-2008-004, NVIDIA (2008)
3. Lawlor, O.S.: Message Passing for GPGPU Clusters: cudaMPI. In: IEEE Cluster PPAC Workshop (2009)
4. Rupp, K., Rudolf, F., Weinbub, J.: ViennaCL - A High Level Linear Algebra Library for GPUs and Multi-Core CPUs. In: Proceedings International Workshop on GPUs and Scientific Applications (GPUScA), pp. 51–56 (2010)
5. Rupp, K., Weinbub, J., Rudolf, F.: Automatic Performance Optimization in ViennaCL for GPUs. In: Proceedings Parallel/High-Performance Object-Oriented Scientific Computing Workshop, POOSC (2011)
6. Tomov, S., Dongarra, J., Baboulin, M.: Towards Dense Linear Algebra for Hybrid GPU Accelerated Manycore Systems. Parallel Computing 36, 232–240 (2010)
7. Zee, F.G.V., et al.: The libflame Library for Dense Matrix Computations. Computing in Science and Engineering 11, 56–63 (2009)
8. AMD Accelerated Parallel Processing SDK, `http://developer.amd.com/gpu/amdappsdk/`
9. Boost uBLAS, `http://www.boost.org/libs/numeric/ublas/`
10. Eigen, `http://eigen.tuxfamily.org`
11. GotoBLAS2, `http://www.tacc.utexas.edu/tacc-projects/gotoblas2/`

12. Khronos OpenCL, http://www.khronos.org/opencl/
13. libflame, http://z.cs.utexas.edu/wiki/flame.wiki/libflame/
14. NVIDIA CUDA, http://www.nvidia.com/cuda/
15. SimuNova Matrix Template Library 4, http://www.simunova.com
16. ViennaCL, http://viennacl.sourceforge.net