# Towards Performance-Portable, Scalable, and Convenient Linear Algebra

Philippe Tillet
*Institute for Microelectronics*
*TU Wien*

Karl Rupp
*MCS Division*
*Argonne National Laboratory*

Siegfried Selberherr
*Institute for Microelectronics*
*TU Wien*

Chin-Teng Lin
*Institute of Electrical and Computer Engineering*
*National Chiao Tung University*

## Abstract

The rise of multi- and many-core architectures also gave birth to a plethora of new parallel programming models. Among these, the open industry standard OpenCL addresses this heterogeneity of programming environments by providing a unified programming framework. The price to pay, however, is that OpenCL requires additional low-level boilerplate code, when compared to vendor-specific solutions, even if only simple operations are to be performed. Also, the unified programming framework does not automatically provide any guarantees on performance portability of a particular implementation. Thus, device-specific compute kernels are still required for obtaining good performance across different hardware architectures.

We address both, the issue of programmability and portable performance, in this work: On the one hand, a high-level programming interface for linear algebra routines allows for the convenient specification of the operations of interest without having to go into the details of the underlying hardware. On the other hand, we discuss the underlying generator for device-specific OpenCL kernels at runtime, which is supplemented by an auto-tuning framework for portable performance as well as with work partitioning and task scheduling for multiple devices.

Our benchmark results show portable performance across hardware from major vendors. In all cases, at least 75 percent of the respective vendor-tuned library was obtained, while in some cases we even outperformed the reference. We further demonstrate the convenient and efficient use of our high-level interface in a multi-device setting with good scalability.

## 1 Introduction

In comparison to the optimization of implementations for traditional single-threaded architectures, the introduction of multi- and many-core architectures has lead to additional degrees of freedom in optimizing implementations for the underlying hardware platforms. In addition, various programming models explicitly targeting many-core architectures such as CUDA, OpenCL, and OpenACC were introduced. This plethora of choices, however, hampers the portability of codes as well as the portability of performance to a much larger degree than with traditional single-threaded architectures.

To address the portability of code, the Open Computing Language (OpenCL) [1, 2] was introduced as an open standard and is intended to provide a common programming model for devices from all major vendors. In general, however, the performance of a particular OpenCL code varies significantly, both between hardware from different vendors as well as between different hardware generations of the same vendor. Since optimizing a compute kernel for a specific architecture requires a thorough understanding of the underlying hardware, a systematic common tuning approach is clearly preferable over manual tuning, both from a productivity and from a maintainability point of view.

Our approach for addressing performance portability is based on an automatic generation of compute kernels from common code templates equipped with several parameters. Such an approach was successfully used in other application domains such as signal processing [3] and is also central in the linear algebra library ATLAS [4], which is restricted to CPUs. Auto-tuning for GPUs has been applied in previous work for a given GPU architecture. Examples are optimizations of matrix-matrix multiplications on NVIDIA GPUs [5] and

AMD GPUs [6]. We extend the techniques used therein to obtain portable performance on multiple, possibly heterogeneous devices for a large set of linear algebra operations. This includes common vector operations such as addition and dot products (level 1), matrix-vector products (level 2) and matrix-matrix products (level 3) as defined in the Basic Linear Algebra Subprogram (BLAS) standard. Composite operations involving a split of workload and communication between host and device such as triangular solves are not explicitly considered in this work, because they merely rely on calling optimized kernels for matrix-vector and matrix-matrix products internally.

The execution of one or more linear algebra kernels involves the following steps: In the case of multiple compute devices, the operations are decomposed into device-specific tasks. These tasks are managed by a centralized dynamic scheduler described in further detail in Sec. 2, whose role is to balance the workload among a user-provided set of OpenCL devices, possibly from different vendors. A similar scheduling framework, StarPU, was recently incorporated into single- and multi-device operations by other authors [7]. Optimized kernels are generated for each device by a template-based kernel generator described in Sec. 3 and then passed to the OpenCL just-in-time compiler in order to produce an optimized executable. The respective parameters for the kernel generation are queried from a built-in device database, which provides reasonable default values for the particular architecture obtained from the auto-tuning environment described in Sec. 4. If desired, the library user can run the auto-tuning process manually to obtain best performance on the target device.

In addition to the auto-tuning capabilities, this work extends a template-based compute kernel generator [8] to provide a convenient application programming interface. It also includes fast OpenCL compute kernels for CPUs, for which previous optimized implementations even got down to the assembly level [9]. All implementations presented here are freely available as open source in the Vienna Computing Library (ViennaCL) [10].

## 2 The Scheduler

Even though modern GPUs provide high FLOP/s, a frequent limitation often encountered in practice is the relatively small amount of GPU RAM. Moreover, additional factors such as driver limitations may reduce the maximum size of a single buffer to a fraction of the GPU RAM only. On the other hand,

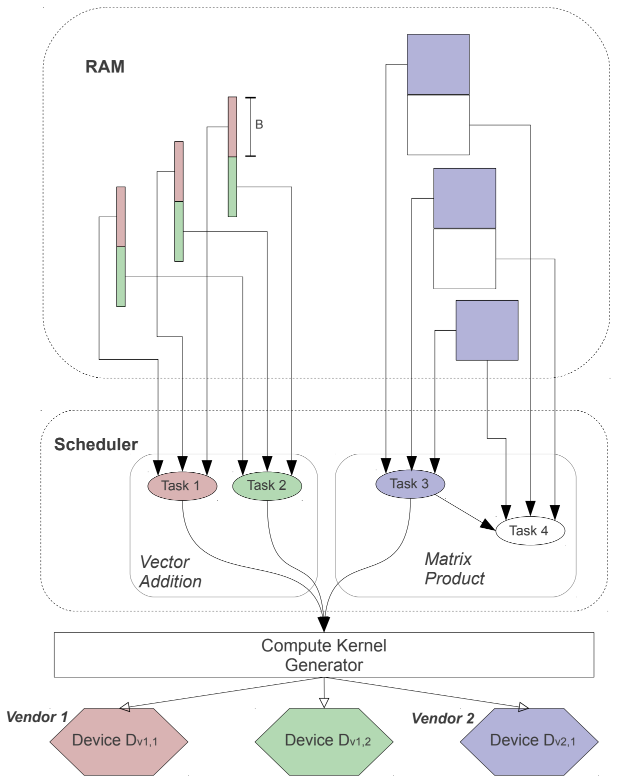

Figure 1: Global execution model employed in this work. In a multi-device setting, memory buffers are split into sub-buffers in the main RAM. The scheduler splits operations on the full objects into operations on smaller objects and forwards these to the device-specific kernel generators.

clever manual memory management is required for the use of multiple GPUs in a single machine. To tackle both difficulties, we employ a centralized, dynamic scheduler which receives the operation specified by the library user and then ensures the efficient execution across multiple devices without exceeding the available memory of each device.

Due to the limited bandwidth of the PCI-Express bus, we impose the commonly used assumption of a compute-limited setting for multiple compute devices in the following. Even though this excludes most sparse linear algebra operations, many popular algorithms such as eigenvalue computations or LU factorizations are covered. In the case that the whole buffer fits into the memory of a single device, it can be directly allocated there and the operations are not further decomposed.

As sketched in Fig. 1, we divide memory buffers which are too large for a single device into chunks stored in the main RAM. Matrices are decomposed into smaller blocks, while vectors are decomposed

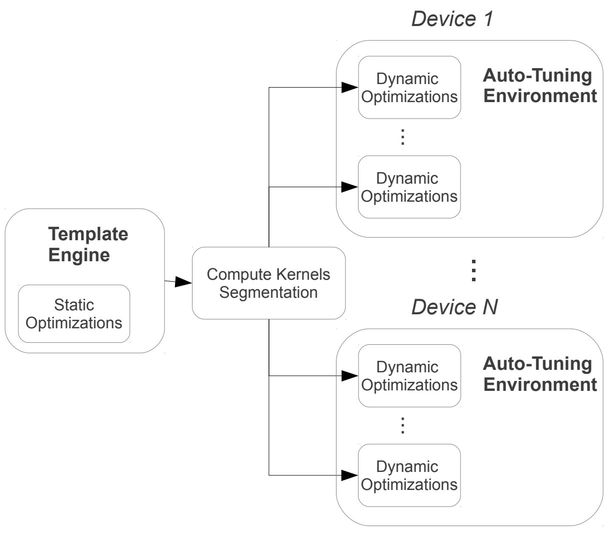

Figure 2: General execution model of the kernel generator. Static optimizations are first applied to the arithmetic operations, and then device-specific optimizations are applied. An optional auto-tuning process can be triggered for each kernel.

into contiguous chunks. As an example, the data in Fig. 1 is partitioned such that the addition of two vectors and the multiplication of two matrices is carried out by three devices. Since these two operations are independent, the scheduler dispatches them such that they are computed concurrently by the three devices.

To achieve a high load on all available devices, we convert each BLAS operation to a task graph [11] acting on the various sub-buffers. The execution and dynamic load balancing [12] is then carried out by the scheduler by forwarding each task to the kernel generator. The kernel generator discussed in the next section ensures that optimized kernels are used on each device, cf. Fig. 2.

## 3 A Template-Based Kernel Generator

We now turn to the description of the kernel generator API and consider as an introductory example the matrix operations

$$U = A + B, \quad V = A - B \tag{1}$$

for matrices $A$, $B$, $U$, and $V$ for execution on a certain compute device. We note that $A$, $B$, and $U$ can also be non-overlapping submatrices of larger, not necessarily distinct matrices $A'$, $B'$, and $U'$. Using C++ operator overloading, these operations translate to code using the kernel generator facility as:

```
1 viennacl::generator::custom_operation op;
2 op.add(U = A + B);
3 op.add(V = A - B);
```

Lines 2 and 3 register the operations at the generator, but do not yet lead to a generation of kernel source code. Instead, the operations are stored in their symbolic form using expression trees in order to allow for the reuse of kernels. Thus, when the execution of an operation set is triggered via op.execute(), a look-up is performed in the OpenCL backend first. If suitable kernels have already been compiled previously, they are retrieved and directly executed. Otherwise, the source code generation process is triggered.

The first step in the kernel generation process is a kernel segmentation step. Here, the data dependencies are scanned and operations possibly reordered without changing the results of the operations This enables maximizing the number of consecutive operations using the same kernel template. For memory bandwidth-limited operations, operations are fused together into a single kernel in order to minimize memory transfers by enabling data reuse. Since such a packing of operands also increases the register pressure and local memory usage on the device, the fusing of multiple statements into a single kernel is not performed for compute-bound operations such as matrix-matrix multiplications. In such a setting, the costs of temporaries is relatively low and usually compensated by the higher occupancy and thus performance of the device for the compute-bound kernel.

Then, the sources for each kernel are generated from the symbolic representation based on kernel templates and their optimization parameters. The latter may be loaded from a user-provided XML file, which can be generated through an auto-tuning procedure and easily shared amongst different users. If no such file is provided, our backend uses a fallback to a built-in, vendor-specific database using reasonable default parameters for each of the different computing architectures.

For the example in (1), the two operations are found to be independent and thus reduced to a single kernel and computed in parallel. This allows to execute both operations using only a single load of the data in $A$ and $B$ from global memory. The kernel is then generated based on the template for the SAXPY operation defined in BLAS. A sketch of the code generation step, the structure of which breaks down to three distinct parts, is as follows:

1. First, optimization directives or preprocessor information is written. Typical examples are

OpenCL pragmas for enabling double precision (if provided by the hardware), or specific hints about the work group sizes the kernel is launched with.

2. In the second step, the operation list is parsed and the kernel function header is created based on the uniqueness of the associated memory handles. For the example in (1), a kernel with four arguments associated with $U$, $A$, $B$ and $V$ is created, rather than a kernel with six parameters consisting of three arguments for each individual computation.

3. Finally, the kernel body is generated. Data from operands on the right hand side of the expression, i.e. $A$ and $B$ in the example above, is loaded into private memory first to eliminate unnecessary global reads. Then, the expression described by the symbolic representation is directly translated into operations in private memory. Suitable tiling as well as vector data types are used in order to obtain structured memory accesses with highest bandwidth.

## 4 The Auto-Tuner

As already indicated in the previous section, the actual kernel generation is parameterized by various hardware-specific parameters. Typical parameters stem from hardware architecture characteristics such as cache sizes or the layout of memory channels and are supplemented by the global and local work sizes for the actual execution. Since an accurate derivation of the best kernel parameters for each device available on the market is practically infeasible, we automize the search for best parameters by a dedicated tuning facility.

The auto-tuning procedure's tractability and performance is largely dependent on the size and bounds of the parameter space associated with the corresponding kernel template. We employ a straight-forward brute-force search based on essentially empirical parameter sets for the various operation templates. This typically leads to total execution times for the tuning process ranging from a few seconds for operations at BLAS level 1 to a few hours in the case of operations from BLAS level 3. Kernels incompatible with constraints given by the maximum work group size CL_KERNEL_WORK_GROUP_SIZE as well as kernels not compatible with the preferred work group size multiple CL_KERNEL_PREFERRED_WORK_GROUP_SIZE_MULTIPLE are not considered in the tuning process, since they would either fail or are a-priori known to map poorly onto the target hardware.

While auto-tuning is clearly impractical if required to be run by all library users, it gives library developers valuable insight into the location of sweet spots for a certain hardware generation. Consequently, auto-tuning is a vital tool for developers in order to provide fast default kernels for each of the available hardware generation so that almost optimal performance is obtained even if the library user is not interested in running an auto-tuner. At the same time, users interested in utmost performance can still run the auto-tuning environment to obtain the last few percents of speed. To do so, auto-tuning executables for BLAS levels 1, 2 and 3 are provided as well as generic functions that allow to run the auto-tuning process even for kernels involving multiple operations, where the limited amount of registers and local memory may lead to different optimal kernels. In this way, also the data dependency of the optimal kernel parameters is addressed, as users can run the tuning process for the particular vector and matrix dimensions used in their applications.

## 5 A Case Study : Performance-Portable Matrix-Matrix-Multiplication Kernels

The parameter space for kernel generation for the BLAS level 1 operations in (1) is small, because only a single operation is carried out for each floating point number and thus no benefit from caching is possible. In contrast, operations at BLAS level 2 and 3 rely on data reuse for good performance and thus lead to a much larger set of parameters. The archetypical example is the GEneralized Matrix-Matrix multiplication (GEMM)

$$C \leftarrow \alpha \times A \times B + \beta \times C\ , \tag{2}$$

with scalars $\alpha$ and $\beta$ and matrices $A$, $B$, and $C$ of dimensions $M \times K$, $K \times N$, and $M \times N$, respectively. In addition to being a fundamental building block in linear algebra, its compute intensive nature makes it a popular algorithm for hardware benchmarks. Optimization efforts have been spent for decades and recent performance studies for CPUs [9], NVIDIA GPUs [5], and AMD GPUs [6] are popular. To achieve high performance on the target device, these implementations rely heavily on well known, yet architecture dependent, blocking optimization techniques. While recent work has considered portability of auto-tuned implementations within hardware

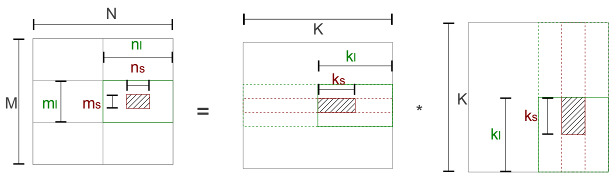

Figure 3: Block computation performed by the work unit of global id (3,4), and group id(1,1)

from a single vendor, we demonstrate that our parameterized kernels also lead to portable high performance among different vendors.

Recent GPUs now include a hardware L1 cache in each of their compute units. As a consequence, state-of-the art GEMM kernels rely on a double-blocking strategy, depicted in Fig. 3. In this configuration, the matrices $A$ and $B$ are divided into large blocks of size $m_l \times k_l$ (resp. $k_l \times n_l$), which are further divided into smaller blocks of size $m_s \times k_s$ (resp. $k_s \times n_s$). Each work-unit of coordinates $(g_i, g_j)$ is then responsible of the computation of the submatrix

$$C[g_i m_s : g_i m_s + m_s - 1, \quad g_j n_s : g_j n_s + n_s - 1] \, .$$

This enables each work-unit to benefit from data fetched by its neighbors, which leads to a drastic reduction of the global L1 miss rate when the block sizes are well chosen.

GPUs integrate local memory to allow the work-units of the same work-group to share data. While this software-managed memory can have a bandwidth twice as high as that of the L1 cache on some devices, it is subject to conflicts, when different work-units fetch data from the same memory bank. Besides, allocating too much local memory limits the number of work groups which can concurrently execute on a given compute unit. Therefore, the proper use of local memory for the large blocks of size $m_l \times k_l$ and $k_l \times n_l$ is an additional degree of freedom in the generator and consequently in the tuning process.

In addition to various memory domains, the OpenCL standard offers a broad range of built-in vector types and is thus more explicit with respect to vectorization than for example the C programming language. Using these vector types properly can lead to higher occupancy rate, or better assembly code, when the platform's compiler translates them for instance to SSE or AVX instructions for CPUs. As a consequence, the use of vector types is an important parameter for the generated kernel's performance and constitutes another degree of freedom for the generator.

In summary, our kernels reside in a nine-dimensional space, whose bounds are empirically chosen. We have found that the parameter space determined by

| | | |
|---|---|---|
| $m_l, k_l, n_l$ | $\in$ | $\{2^k, 5 \le k \le 8\}$ |
| $m_s, k_s$ | $\in$ | $\{2^k, 1 \le k \le 3\}$ |
| $n_s$ [GPU] | $\in$ | $\{2^k, 1 \le k \le 3\}$ |
| $n_s$ [CPU] | $=$ | $n_l$ |
| alignment | $\in$ | $\{2^k, 0 \le k \le 3\}$ |
| fetch from A to local memory | $\in$ | {true, false} |
| fetch from B to local memory | $\in$ | {true, false} |

leads to tractable auto-tuning procedures. It should be noted that even though we restrict our parameters to be powers of two, our parameter space consists of 27 648 kernels for one operation in the case of GPUs Additional tuning runs for operations involving either transposes different memory layouts (row-major versus column-major) are required. Moreover, optimization studies of matrix-matrix multiplications for NVIDIA GPUs suggest that $m_l$, $k_l$, and $n_l$ are better chosen to be multiples of 16 rather than powers of two [5], leading to a further increase of the search space. Nevertheless, even a restriction to powers of two leads to high portable performance across different CPU and GPU vendors as shown next.

## 6 Examples and Results

Our benchmarks compare the performances obtained for three different architectures, namely an INTEL Core i7 960 CPU, an NVIDIA Geforce GTX 470, and an AMD Radeon HD 7970. The hardware is chosen to represent a broad range available in average desktop machines rather than high-end configurations only. As a performance reference, the implementations in Intel MKL 11.0, CuBlas 5.0, and clAmdBlas 1.10 were used, respectively. Even though our generator can deal with both row- and column-major matrices, the matrices considered for the benchmarks were taken to be all row-major for simplicity.

We first review the bandwidth-limited operation

$$\beta \leftarrow x^T \times (2.x + y) \qquad (3)$$

for vectors $x$, $y$, and scalar value $\beta$ on a single compute device. While this operation requires two distinct BLAS level 1 calls for each of the vendor libraries, (3) is parsed as a single operation by our generator, which enables the operation to be computed loading the data in $x$ and $y$ only once. Fig. 4

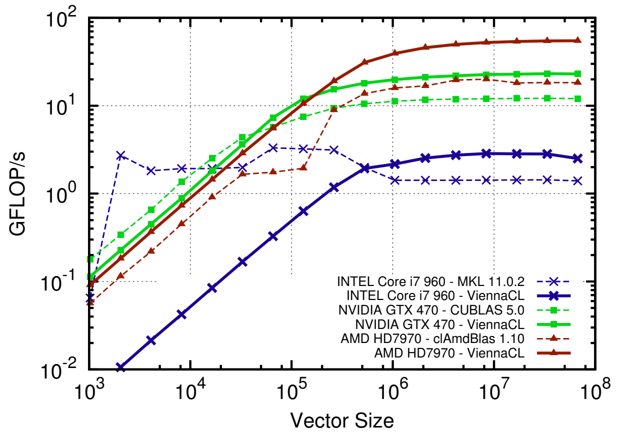

Figure 4: Performance of $\beta = x^T \times (2.x + y)$ in double precision.

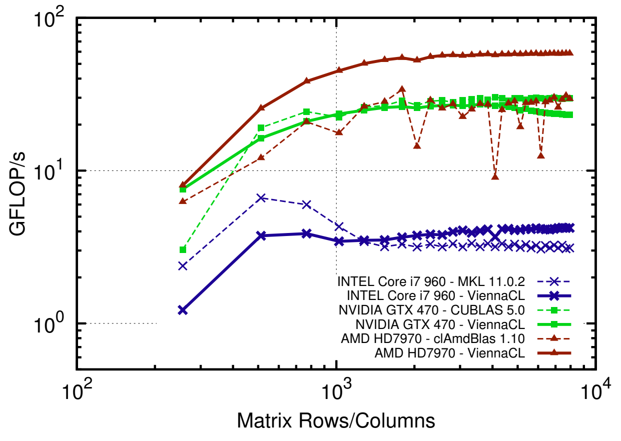

Figure 6: Comparison of DGEMV performance

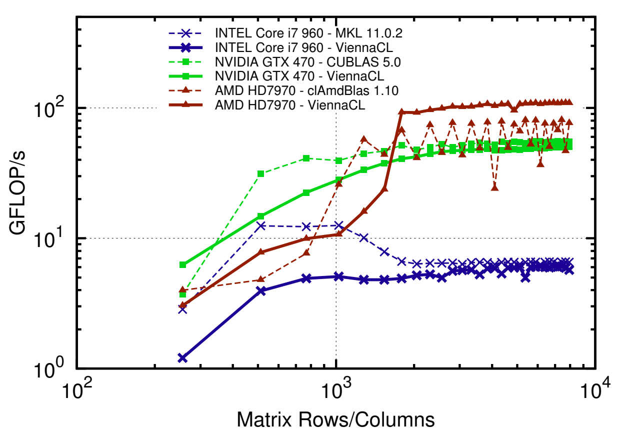

Figure 5: Comparison of SGEMV performance

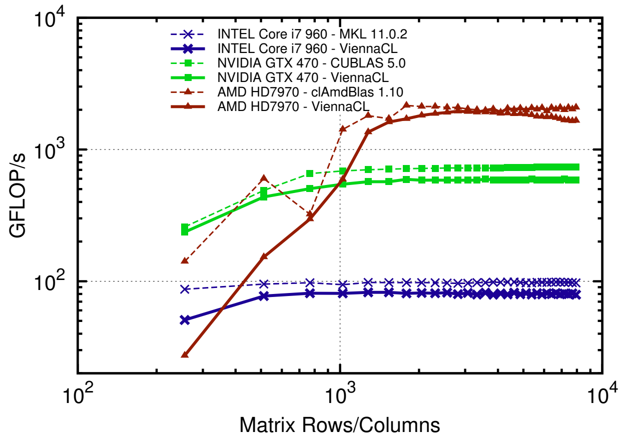

Figure 7: Comparison of SGEMM performance.

shows that such memory optimization leads to a gain of a factor of two at large vector sizes over the reference implementations on all three devices considered. For small sizes, management overhead in PCI-Express communication and the OpenCL backend becomes apparent, the latter particularly on the CPU. Our results further show that the BLAS interface is only poorly suited for such a heavily memory bandwidth-limited setting, because data reuse across multiple BLAS calls other than through cache is impossible. Since the results for single precision exhibit exaclty the same qualitative behavior, we omit these for the sake of brevity.

Our second benchmark evaluates the performance of our framework for the GEneral Matrix-Matrix multiplication (GEMV) routine, i.e. the operation

$$y \leftarrow \alpha \times A \times x + \beta \times y$$

for a matrix $A$, vectors $x$, $y$, and scalar values $\alpha$, $\beta$. The results are depicted in Fig. 5 for single precision (SGEMV) and in Fig. 6 for double precision (DGEMV). In all the cases, our best kernels exhibit performance ranging from 80 percent to 200 percent of those obtained using vendor-tuned libraries. The poor performance on small problems for the Intel CPU is attributed to the overhead induced by the OpenCL backend. However, for problems of size greater than 1024, our double precision implementation achieves better performance than the Intel MKL. We also outperform clAmdBlas, for which the vendor-provided auto-tuning framework was used, in both the SGEMV and the DGEMV case.

A comparison of the obtained performance for the GEMM operation (2) considered in our case study in Sec. 5 is given in Fig. 7 for single precision (SGEMM) and Fig. 8 for double precision (DGEMM). In all six cases our generated compute kernels obtain at least 75 percent of the peak performance of vendor-tuned libraries. Our approach outperforms the tuned

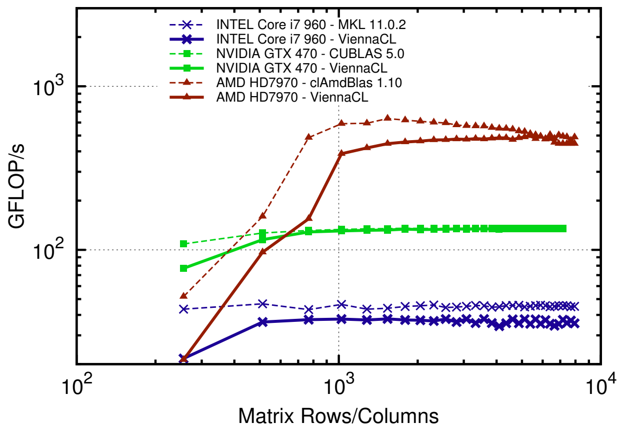

Figure 8: Comparison of DGEMM performance.

clAmdBlas library in the SGEMM case at big problem sizes and is on par with CUBLAS 5.0 for the DGEMM case. The performance difference in the case of SGEMM on the NVIDIA GPU is due to the choice of our parameter space, where other authors have shown that best performance is actually obtained for block sizes not being powers of two [5].

In our last example we demonstrate the flexibility and performance of our framework based on the operation

$$C \leftarrow (A+B)\times(A-B)\ . \tag{4}$$

on a system with two NVIDIA GPUs in single precision. In order to execute this operation concurrently on all available GPUs, we provide the following high level API:

```
add_all_available_devices(CL_DEVICE_TYPE_GPU);
multi_matrix<float> C, A, B;
/* Fill matrices with data here */
C = prod(A + B, A - B);
finish();
```

Here, all available devices are first attached to the scheduler. The multi_matrix<> type explicitly creates matrix objects for use on multiple devices. The operation is then triggered and the main process stalled until completion.

Using operator overloads and symbolic representations, our engine is able to deal directly with matrix expressions instead of temporary matrices. However, since performing such calculations inside the GEMM kernel would significantly increase the register pressure, we compute temporaries at the block level, which guarantees a negligible memory consumption overhead in practice without detrimental effect on overall performance. As shown in Fig. 9, our approach considerably reduces the total memory consumption, which enables handling

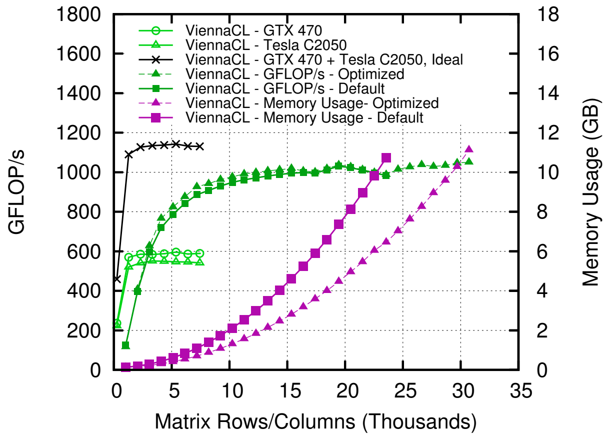

Figure 9: Performance of $C = (A+B)\times(A-B)$ for a machine equipped with an NVIDIA GTX 470 and an NVIDIA Tesla C2050 in single precision. The optimized case refers to temporaries on the block level, while the default case uses explicit temporaries for $A+B$ and $A-B$.

problem sizes up to 31 000 instead of 24 000, when using full temporaries for $A+B$ and $A-B$ on a machine with 12 GB main memory. An overall performance of 90 percent of the ideal case of summing the two single-GPU-performances is obtained.

## 7 Outlook and Conclusion

As with all auto-tuning approaches, a weakness of our framework is the empirical choice of bounds for the parameter space, still leading to a large search space. For this reason, future work lies in reducing the time spent on auto-tuning or widening its exploration space by using clever adaptive strategies. Even though a database for storing and retrieving devices allows to provide close-to-peak performance for library users on a wide range of hardware without necessarily exposing them to the auto-tuning process, the plethora of different hardware available makes it nevertheless hard for library implementors to provide sufficient coverage.

In summary, our framework addresses issues related to performance portability in heterogeneous environments by proposing dynamic kernel generation to tackle this issue. Comparable or even better performance than vendor-provided libraries is obtained for the three BLAS levels. The convenient use of multiple devices was addressed using work partitioning, scheduling, and operator overloads, while scalability is achieved using a temporary removal mechanism.

## References

[1] OpenCL. `http://www.khronos.org/opencl/`.

[2] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From CUDA to OpenCL: Towards a Performance-Portable Solution for Multi-Platform GPU Programming. *Parallel Computing*, 38:391–407, 2012.

[3] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy Johnson, David Padua, Manuela Veloso, and Robert W. Johnson. Spiral: A generator for platform-adapted libraries of signal processing algorithms. *Int. J. High Perform. Comput. Appl.*, 18(1):21–45, 2004.

[4] ATLAS Library. `http://math-atlas.sourceforge.net/`.

[5] J. Kurzak, S. Tomov, and J. Dongarra. Autotuning GEMM Kernels for the Fermi GPU. *IEEE Transactions on Parallel and Distributed Systems*, 23(11):2045–2057, 2012.

[6] K. Matsumoto, N. Nakasato, and S. G. Sedukhin. Implementing a Code Generator for Fast Matrix Multiplication in OpenCL on the GPU. In *6th IEEE International Symposium on Embedded Multicore SoCs (MCSoC-12), 2012*, pages 198–204, 2012.

[7] E. Agullo, C. Augonnet, J. Dongarra, H. Ltaief, R. Namyst, S. Thibault, and S. Tomov. A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs. In *GPU Computing Gems*, volume 2, pages 473–484. Morgan Kaufmann, 2010.

[8] Ph. Tillet, K. Rupp, and S. Selberherr. An Automatic OpenCL Compute Kernel Generator. In *Proceedings of the 2012 Symposium on High Performance Computing*, 2012.

[9] K. Goto and R. A. van de Geijn. Anatomy of High-Performance Matrix Multiplication. *ACM Transactions on Mathematical Software*, 34(3):125, 2008.

[10] Vienna Computing Library (ViennaCL). `http://viennacl.sourceforge.net/`.

[11] F. Song, A. YarKhan, and J. Dongarra. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 19:1–19:11. ACM, 2009.

[12] O. Villa, S. Krishnamoorthy, and G. R. Gao. Dynamic Load Balancing on Single and Multi-GPU Systems. In *IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12, 2010.