URN (Paper): urn:nbn:de:gbv:ilm1-2014iwk-112:9

58th ILMENAU SCIENTIFIC COLLOQUIUM
Technische Universität Ilmenau, 08 – 12 September 2014
URN: urn:nbn:de:gbv:ilm1-2014iwk:3

A SELF-OPTIMIZING FRAMEWORK FOR DEVELOPING METROLOGY SOFTWARE ON MASSIVE PARALLEL PROCESSOR ARCHITECTURES

Beier, T.[1]

[1] Carl Zeiss Jena GmbH, Development and Projects, Development Optical Systems, Carl-Zeiss Promenade 10, 07745 Jena, Germany

ABSTRACT

Standard PC hardware rapidly increases in parallel computing power in form of multicore CPUs and general purpose GPUs. To take advantage of this situation it is necessary to create specialized code. This is a very time consuming and therefore an expensive task. One approach on solving this problem is the OpenCL (Open Computing Language) standard. It offers the possibility to run the same code on different hardware platforms. OpenCL provides code portability but not performance portability. This paper introduces the concept of a new developed self-optimizing parallel programming framework that addresses the issue of performance portability. This framework provides a set of algorithm building blocks. With the help of these building blocks a wide range of algorithms can be described, that work on one- or two-dimensional objects like images, vectors and matrices. The achieved performance is demonstrated with different algorithms on standard computer hardware platforms.

1. INTRODUCTION

The precise measurement of optical surfaces requires not only high performance measurement instruments but also extreme compute intensive algorithms. Production constrains often dictate that measurement tasks should be finished as fast as possible. In the last decade it became clear that the increase in computing power of CPUs by increasing the clock rate could not be kept. In 1965 Moore predicted that the count of transistors, that can be incorporated into integrated circuits, will double every year [1]. This timespan has later been corrected to 18 month, but the prediction still holds. The increase in transistor count and the flattening in clock rate gain lead to a dramatic increase of parallelism in computer architectures over the last decade. Nowadays CPUs that are used in standard PCs have at least two or more cores. Each of them supports parallel SIMD instruction sets like SSE or AVX. That enables the usage of SIMD commands. SIMD is a classification for computer architectures in Flynn's taxonomy and stands for single instruction, multiple data (-streams) [2]. It says that a single instruction is executed on multiple data-streams in parallel. For example the AVX2 extension supports eight element wide single precision SIMD instructions. A single Intel® Haswell CPU core can execute two AVX2 FMA instructions in parallel. This enables a throughput of 32 single precision floating point operations per cycle and 16 double precision floating point operations per cycle and core [3]. This shows that CPUs became massive parallel processors.
Another development is the introduction of dedicated Graphic Processing Units (GPUs) that originally handled the calculation of complex graphical scenes in two and three dimensions. GPUs became more and more powerful massive parallel processors over time. Nowadays they are able to handle not only image processing tasks but also general purpose computations. Meanwhile GPUs are widely used as coprocessors for intensive parallel computations. One example is the number 2 in the latest TOP500 list [4]. This supercomputer, called "Titan", uses 18.688 NVIDIA® Tesla C2050 Cards that are based on the NVIDIA® Fermi GPU

architecture. The currently most powerful supercomputer, the Tianhe 2, uses 48.000 Xeon Phi coprocessors. The Intel® Xeon Phi consists of 57 processor cores and was originally designed as graphic processing architecture. Now it serves as massive parallel coprocessor for numerical intensive tasks. These developments lead to the point that today nearly every standard PC not only contains a multicore processor, but also a massive parallel GPU. This can be used for general purpose computations. At first view it seems quite reasonable to exploit the parallel computing power of both CPU and GPU for computation intensive tasks. At closer look development of software that runs on GPUs and other massive parallel hardware is a very demanding and time consuming task.
For the development of GPU based programs many programming interfaces are available. Two of the most important are CUDA and OpenCL. CUDA is developed by NVIDIA and runs exclusively on hardware of this manufacturer. In contrast, OpenCL is supported by numerous hardware manufacturers like AMD, Intel, ARM and also NVIDIA. OpenCL programs can be executed on any hardware that provides an OpenCL driver. Based on this flexibility, OpenCL seems to be a good choice for the usage as programming interface for the development of portable and efficient software. OpenCL programs have to be optimized for specific hardware architectures to run efficiently on them. This shows that it provides portability of code between hardware architectures, but cannot ensure portability of performance. This work introduces a framework that has been designed to overcome the problem of performance portability. It enables the creation of parallel numeric algorithms that can be run on nearly any OpenCL supporting hardware, while maintaining considerable performance.

2. OPENCL

OpenCL is an open standard for parallel programming of parallel processors like CPUs, GPUs and other computer architectures. It is developed by the Khronos Group. The latest version is 2.0 but there is currently no implementation available. This work will focus on version 1.2 [5]. OpenCL provides a platform API for managing and executing OpenCL programs that are called OpenCL kernels. OpenCL kernels can be executed on any hardware that delivers OpenCL drivers. Currently there exist OpenCL implementations from nearly every big hardware vendor. Some of them and their supported hardware architectures are listed in Table 1. OpenCL is based on an abstract hardware platform model that is shown in Figure 1.

Vendor	Architectures
Intel	CPU, integrated GPU, accelerator
AMD	CPU, GOU, integrated GPU
NVIDIA	GPU
IBM	CPU, CELL

Table 1: Hardware vendors and their computer architectures which are currently supporting OpenCL

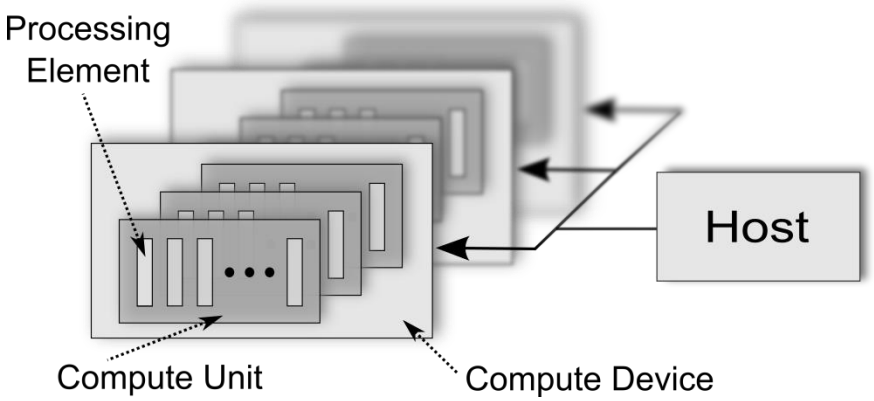

Figure 1: The OpenCL platform model

The host of an OpenCL environment runs the application program. It uses OpenCL API calls to manage the creation and execution of OpenCL programs on one or more compute devices that map accessible compute resources like CPUs or GPUs. A compute device is separated into Compute Units that bundle a number of processing elements.
OpenCL kernels are executed on a predefined discrete index space that can have up to three dimensions. For every point in that index space, called NDRange, an instance of the kernel is executed. The index space is clustered into workgroups. An example of a two dimensional index space is shown in Figure 2.

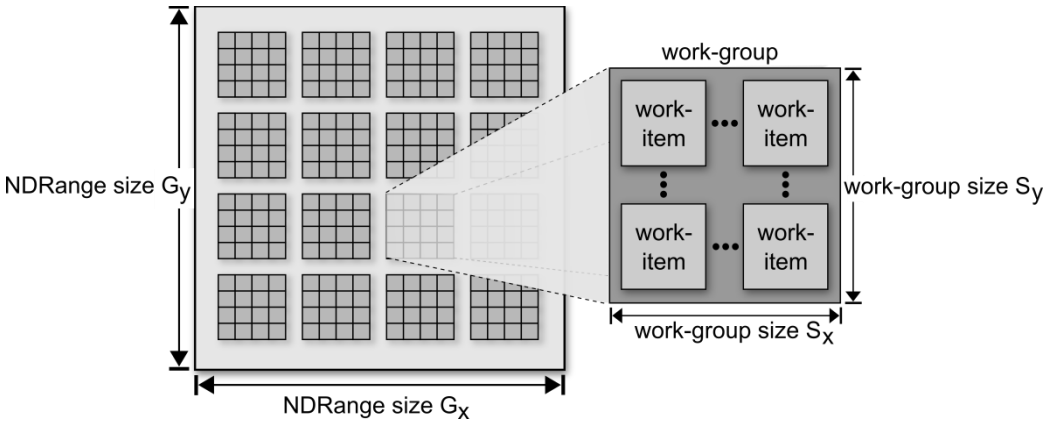

Figure 2: The OpenCL execution model for a two dimensional index space

OpenCL kernels are written in the OpenCL C language. It is a subset of the C99 standard and has been expanded with vector data types, vector instructions and OpenCL mechanic specific functions.

3. THE FRAMEWORK

3.1 Introduction to the framework

The presented framework enables the creation of numerical libraries. Its purpose is to reach reasonable performance on different hardware architectures without specific optimizations in the definition of the programs. To achieve this goal the framework relies on OpenCL as target backend. The usage of OpenCL enables the framework to run on nearly every device that supports at least Version OpenCL 1.1. Therefore the framework is designed for single

OpenCL hosts and the usage of a single OpenCL device. The coarse design idea of the proposed framework is outlined in Figure 3.

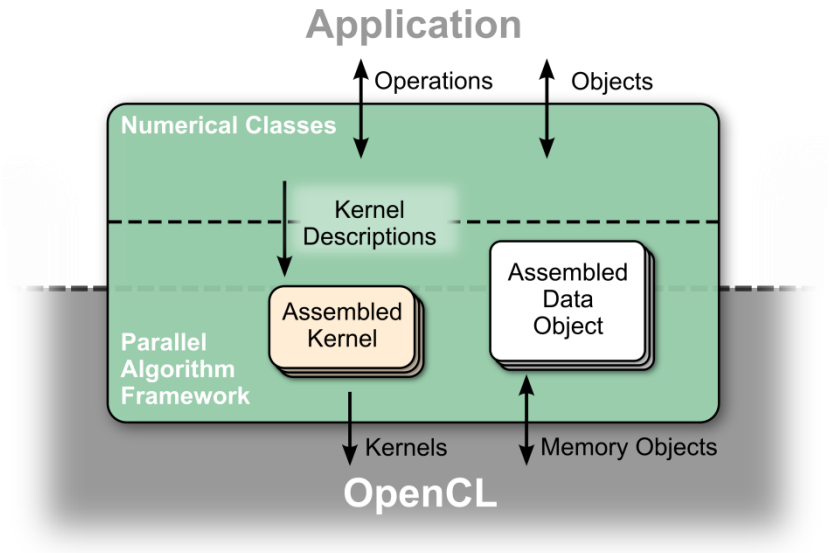

Figure 3: Coarse architecture view of the proposed framework

The framework is split into three different levels. On the application side, a number of numerical object classes enable an application to create numerical objects like matrices and vectors with different element types. The numerical classes define operations that can be applied on numerical objects. The operation kernels are defined in a description language that is presented in chapter 3.2. Data transfers between the application and OpenCL devices are automatically managed and minimized.

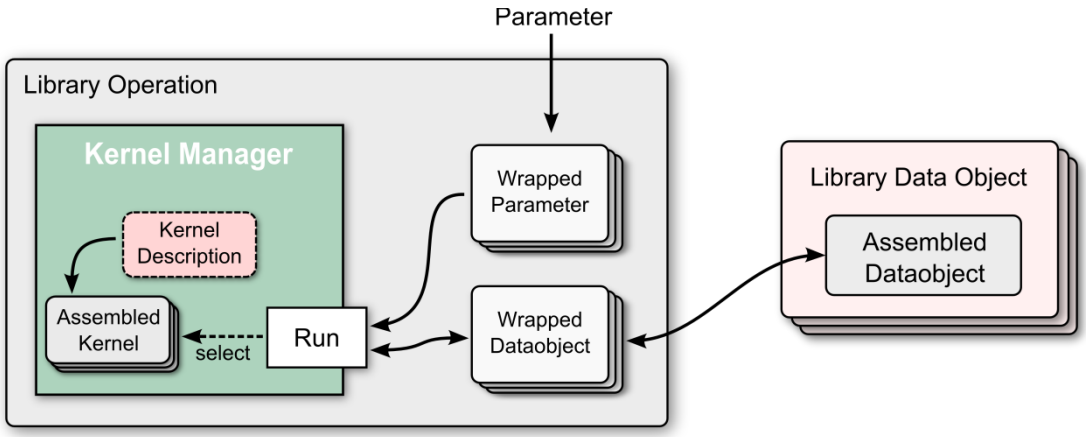

Figure 4: Operations of the numerical library mainly serve as wrapper for the execution of predefined kernels. Those kernels are managed by a class called kernel manager.

To simplify the creation of kernels, a management class is introduced. It is a class prototype for easy management of kernels and represents the second level of the framework. Its usage scenario is shown in Figure 4. The root kernel manager class has to be overloaded to create a specific kernel manager class. In the overloading process it is necessary to implement functions that deliver a kernel description and sets of benchmark descriptions that are used to create different data objects for benchmark purposes.

The third level is nearest to the OpenCL side. It consists of the Assembled Kernel and the Assembled Data Object. The Assembled Kernel consists of a compiled OpenCL kernel and all information needed to execute it. The Assembled Data Object consists of a pointer to an OpenCL memory object and information how to handle it.
To adopt the defined kernels to the underlying hardware a tuning step is performed. It is outlined in Figure 5. In a first step the kernel manager creates different Assembled Kernels. Those kernels differ in code variation and their parallel distribution schemes. Kernel variations are described in chapter 3.4. In the next step the kernel manager creates a set of data objects and parameters for every defined benchmark set. In an auto-tuning process the kernel manger searches the Assembled Kernel that runs fastest for the data objects and parameters of a specific benchmark set. This results in a benchmark tuple that connects a reference to the fastest kernel with a benchmark set.

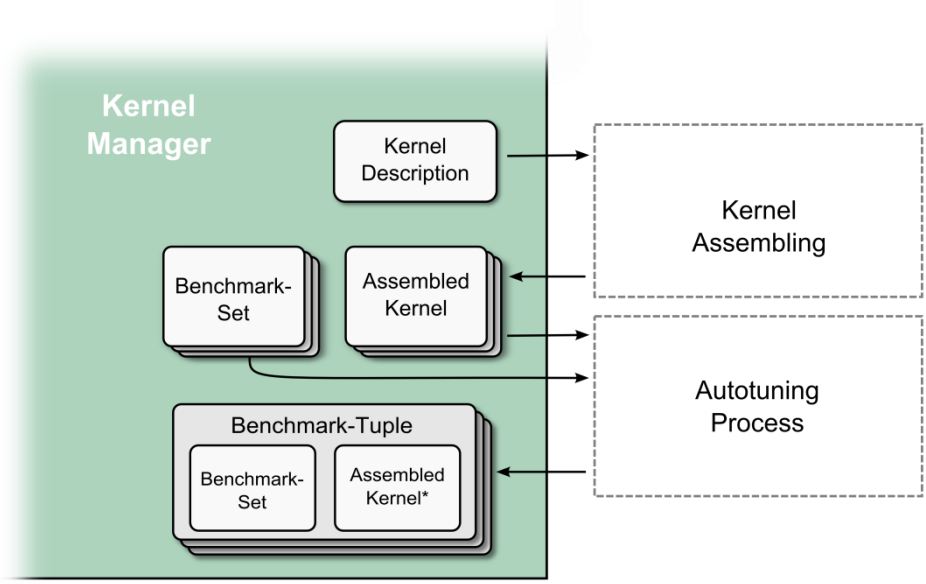

Figure 5: Auto-tuning Process

The library can execute the kernel by calling the "Run" routine of the overloaded kernel manager class at runtime. It has to provide a list of wrapped parameters and a list of wrapped data objects that create a connection between Assembled Data Objects and the data object names used in the kernel description. The kernel generator decides which assembled kernel is used from the given parameters, the saved benchmark tuples and the properties of the wrapped data objects. The benchmark process has to be executed only once per specialized kernel manger and OpenCL device. The resulting benchmark tuples are saved in a configuration file and can be loaded within the initialization process of the library.

3.2 The description language

A description language has been developed for this framework to describe parallel numerical algorithm kernels. It uses an XML-like style and is data-flow oriented. Its development was focused on different aspects:

- Simplicity: Use only a small number of constructs to describe a kernel
- Power: It should deliver the possibility to solve nearly every numerical problem
- Implicit parallelism: The programmer should not be forced to think about parallelism in the algorithms

A description of a parallel algorithm in the framework consists of a number of objects and a list of connections that describe the data-flow between these objects. Every object has connection emitters and connection acceptors that define the direction of data flow. A connection emitter has to be connected to a connection acceptor. The value of the connection emitter is assigned to the connection acceptor at the beginning of the execution of the acceptors object. In the assembling process it is necessary to determine an order of the execution of the objects. This order is detected automatically from the connection graph that is defined by the connections in the description. Optionally the programmer can add a list of ordering rules that consist of a tuple *(object1, object2)* which means that *object1* has to be executed prior to *object2*. These rules are considered in the ordering process.
Some objects can house other objects and create their own iteration space. The iteration space defines the how many times that child objects consume, process and emit data. There is no direct mapping between the number of executions of a kernel body and the number of elements in a data object. To describe the kernels, there are basically six constructs:

- *ParallelAlgorithm*: This object serves as hull of the numerical kernel. Its number of dimensions and "Size(…)" properties define the main iteration space. Optionally it can own an iteration connection function (ICF) that can connect elements of the main Iteration Space and delivers a single data value to a *DataObject*. The general structure of a *ParallelAlgorithm* object is shown in Listing 1:

```
<ParallelAlgorithm name="..." dimension="..." ICF="...">
	<DataObjects>
		...
	</DataObjects>
	<Children>
		...
	</Children>
	<Connections>
		...
	</Connections>
</ParallelAlgorithm>
```

Listing 1: General structure of a ParallelAlgorithm.

- *DataObject*: This Object consists of a number of elements that are ordered in schemes. Currently there are two different organization schemes available: vector and matrix. The element types of a *DataObject* are described in Table 2.

element type	description
SP	single precision floating point number (IEEE 754)
DP	double precision floating point number (IEEE 754)
INT32	32 bit integer format
SP2	2 single precision floating point numbers (IEEE 754) for the representation of complex numbers
DP2	2 double precision floating point numbers (IEEE 754) for the representation of complex numbers
B	boolean value

Table 2: Data types of DataObject elements.

The elements of a *DataObject* can only be changed by *DataAccess* objects.

- *DataAccess*: This object enables a kernel to access the elements of the specified source *DataObject*. It exists in two versions: *DataAccessRead* and *DataAccessWrite*. *DataAccess* objects own a selector function that describes how a number of selection variables define the selected element of the source *DataObject*. The selection variables of a *DataAccess* object are connection acceptors. *DataAccessRead* objects own a connection emitter "out" that delivers the read data value of the selected element of the source object. The connection acceptor "in" of the *DataAccessWrite* object receives the value that has be written to the selected source *Data Object*.

- *FunctionBlock*: This describes a function that connects variables that are connection acceptors and returns the result of this function as a connection emitter.

- *IteratorBlock*: This Object expands the iteration space temporarily with up to two additional dimensions, depending on the number of dimensions and the values of the connection acceptors "Size(…)". An *IteratorBlock* itself can contain objects with the exception of *ParallelAlgorithm* and other *IteratorBlock* objects.

3.3 Example

In this chapter an implementation of the dense matrix-vector multiplication $\vec{c} = A \cdot \vec{b}$ is described. It is the one that is used in the presented framework benchmark in chapter 4. The description code is shown in Listing 2.

```
<ParallelAlgorithm name="MV" dimensions="1">
   <DataObjects>
      <DataObject name="A" structure="matrix" elementDataType="USED_FLOAT"/>
      <DataObject name="b" structure="vector" elementDataType="USED_FLOAT"/>
      <DataObject name="c" structure="vector" elementDataType="USED_FLOAT"/>
   </DataObjects>
   <Children>
      <IteratorBlock name="iter1" dimensions="1" ICF="#acc(0)+#in(0)"
       ICFType="USED_FLOAT">
        <DataAccessRead name="access_A" source="A" selFunction="v0, v1"/>
        <DataAccessRead name="access_b" source="b" selFunction="v0"/>
        <FunctionBlock name="MulFunc" numInVars="2" numOutVars="1"
                       dataType="USED_FLOAT" function="#in(0) * #in(1);"/>
      </IteratorBlock>
      <DataAccessWrite name="access_c" destination="c" accessFunction="v0"/>
   </Children>
   <Connections>
      <Connection source="A.Size(1)" destination="MV.Range(0)"/>
      <Connection source="A.Size(0)" destination="iter1.Range(0)"/>
      <Connection source="iter1.i(0)" destination="access_A.v(0)"/>
      <Connection source="MV.i(0)" destination="access_A.v(1)"/>
      <Connection source="iter1.i(0)" destination="access_b.v(0)"/>
      <Connection source="access_A.out" destination="MulFunc.in(0)"/>
      <Connection source="access_b.out" destination="MulFunc.in(1)"/>
      <Connection source="MulFunc.out(0)" destination="iter1.func(0)"/>
      <Connection source="iter1.out" destination="access_c.in"/>
      <Connection source="MV.i(0)" destination="access_c.v(0)"/>
   </Connections>
</ParallelAlgorithm>
```

Listing 2: Kernel description of dense matrix-vector multiplication

The first step in every kernel is the creation of a *ParallelAlgorithm* object. It is named "MV" in this example (line 2). The next step is to define the used *DataObject*s (lines 2 - 6). With the matrix $A \in \mathbb{R}^{M \times N}$ the operation $\vec{c} = A \cdot \vec{b}$ can be formulated as $c_i = \sum_{k=1}^{M} A_{i,k} \cdot b_k$ for $i = 1\ to\ N$. The Iteration Space *of* "MV" is set to the size of the returning vector $\vec{c}$ that is equal to the number of rows in A. This is achieved by connecting the "Size(1)" attribute of the *DataObject* "A" to the *ParallelAlgorithm* objects attribute "Range(0)" that represents the size of its iteration space dimension 0 (line 18)
To formulate the reduction sum $\sum_{k=1}^{M} A_{i,k} \cdot b_k$ an *IteratorBlock* object called "iter1" is added. It has the ICF "#acc(0) + #in(0)" and a one dimensional iteration space (line 8). The ICF says that the accumulation variable "#acc(0)" is set to the sum of itself and the *ICF* input variable "#in(0)". The size of its iteration space is set to the number of columns in A (line 19).
To perform the multiplication of the elements of A and $\vec{b}$ two reading *DataAccessRead* objects are created. They have selector functions that use unmodified input variables *v0* and *v1* (lines 10-11). The input variables are connected to the iteration spaces of the *ParallelAlgorithm* and the *IteratorBlock.* (lines 20-22). The outputs of the *DataAccessRead* objects are connected to the input variables of a *FunctionBlock* named "MulFunc" that carries out the multiplication (lines 23-24 and 12-13). Its output is connected to the input of the ICF of the enclosing *IteratorBlock.* (line 25)
The calculated sum of the *IteratorBlock* has to be written to the *DataObject* "c", thus a *DataAccessWrite* object named "access_c" is added. Its input variable is connected with the output of the *IteratorBlock*. Its selector function is the unmodified input and its access variables are connected to the current position in the iteration space of the *ParallelAlgorithm* (lines 15 and 26-27).
The used floating point precision can be chosen by replacing the string "USED_FLOAT" with an element data type listed in Table 2.

3.4 Kernel variation

There are some significant benefits from the introduction of a building-blocks-like description language. It creates the possibility to search not only for an optimal OpenCL work group size, but also for possible code variants. This can increase the achieved performance on some hardware architectures. Some of them are:

- Use of shared memory: The selector functions of *DataAccessRead* objects are analyzed and it is decided if it is beneficial to cache the accesses in local memory.

- *SIMDization*: The framework tries to map the defined kernel to OpenCL vector data types and vector operations. This can be beneficial for Architectures like CPUs that directly support SIMD operations.

- Parallel reduction operations: With the ICF of the *ParallelAlgorithm* and the *IteratorBlock* objects exists a direct possibility for the formulation of reduction operations. The framework can choose between different strategies to perform a reduction.

Every object in the description can have code variations. Every possible code variation has different states. At minimum they have two different states: on or off. Some of them like *SIMDization* have more {Off, vector2, vector4,..}.

The other factor that can be varied is the OpenCL workgroup size that is used to execute the created OpenCL kernels. The framework can choose for any of the used $d \in \{1,2\}$ workgroup dimensions D_i from the set $\{1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024\}$. Every OpenCL device has a upper limit E that can be read out from the OpenCL API. The workgroup size is limited by the condition $\prod_{i=1}^{d} D_i < E$. Let $\bar{D}$ be the number of the usable workgroup sizes and $\bar{V}$ of possible code variations, then the complete number of variations $\bar{K}$ of a kernel results from:

$$\bar{K} = \bar{D} \times \bar{V}$$

4. RESULTS

Some kernels have been implemented to prove the ability of the framework to create kernels in different areas of numerical science. They have been benchmarked on three completely different hardware platforms. The used hardware platforms are listed in Table 3.

Architecture	Device Type	Single Precision Peak Performance [Gflop/s]	Double Precision Peak Performance [Gflop/s]	Peak Bandwidth [GByte/s]
Intel® Core i7 960	CPU	102,4	51,2	25,6
NVIDIA® GeForce GTX 480	GPU	1345	168	177
AMD® Radeon HD 7750	GPU	819	51	72

Table 3: Overview of benchmarked hardware architectures

The following kernels have been created and benchmarked:

- *Laplace2D:* Laplace operator of the size 3x3 elements for floating point matrices. For the benchmark a matrix of the size 2048 x 2048 was used.
- *L2Norm:* This kernel calculates the L2-norm of a given vector. For the benchmark a vector of the length of 4194304 elements was used.
- *Max:* This kernel returns the maximum element of a given vector. For the benchmark a vector of the length of 4194304 elements was used.
- *Add:* This kernel adds two matrices. For the benchmark a matrix of the size 2048 x 2048 was used.
- *GeMM:* This kernel multiplies two matrices. For the benchmark a matrix of the size 1024 x 1024 was multiplied with a matrix of the same size.
- *SpMV:* This kernel multiplies a sparse matrix (format: CSR) with a vector. For the benchmark a matrix of the size 524288 x 524288 with 32 Elements per row and a vector of the length of 32768 elements were used.
- *GeMV:* This kernel multiplies a dense matrix with a vector. For the benchmark a matrix of the size 2048 x 2048 was multiplied with a vector of the length 2048.

The reached fractions of either the peak performance or the peak bandwidth for the different kernels have been calculated. The achieved numbers for single precision are shown in Figure 6 and the numbers for double precision are shown in Figure 7.

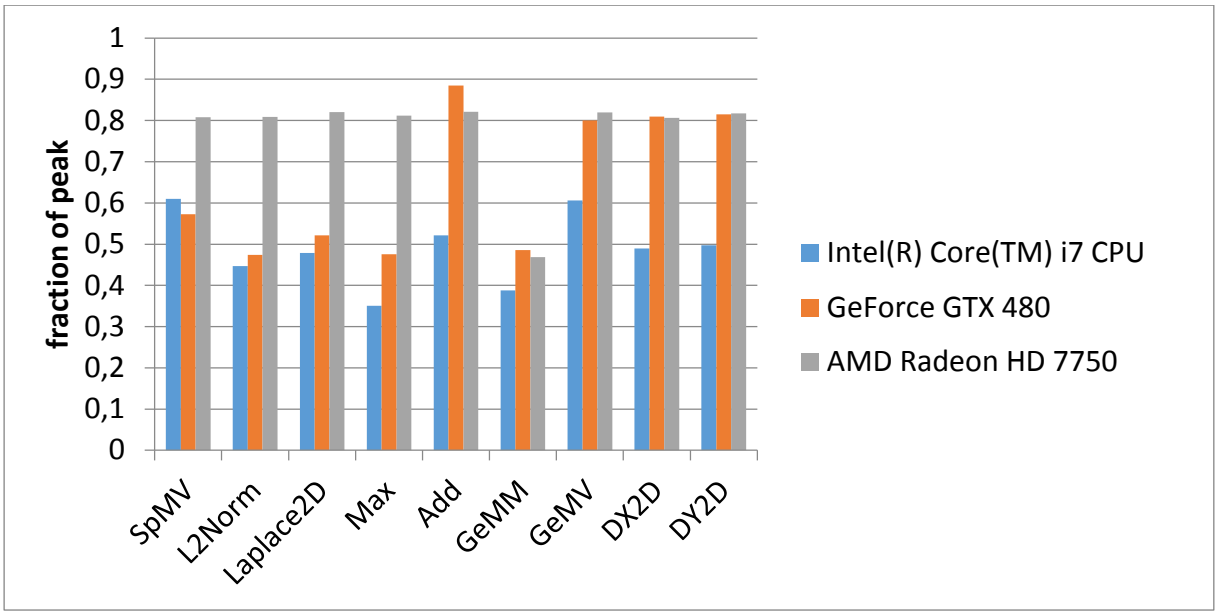

Figure 6: Benchmark of some common single precision kernels that have been implemented in the proposed framework. Shown is the fraction of either the peak performance or the peak bandwidth depending on the limiting factor of the kernel.

Overall the framework delivers considerable single precision performance for the three architectures. The performance on the GPUs never falls below 45% of the peak performance for the corresponding kernel. On the Radeon GPU the framework achieves more than 80% of the peak performance for seven of the eight tested kernels. Sole exception is the GeMM kernel. This kernel is limited by the peak floating point throughput of the GPU and is difficult to optimize. The example GeMM program of the AMD APP SDK 2.9 reaches the same performance as the framework (ca. 380 GFlop/s for the stated matrix sizes).
The performance on the NVIDIA® GTX 480 reaches more than 80% of peak for four of the eight tested kernels. This shows that there is some space for improvement. The GeMM kernel achieves a comparable efficiency as on the AMD® Radeon HD 7750 GPU. Even sophisticated programming and modelling efforts lead to only 63% efficiency for the stated matrix sizes on the NVIDIA® Fermi architecture [6]. From this perspective the reached GeMM performance on the both architectures seems to be within the range of the optimum.
The framework performs at between 35% and 61% of the peak performance on the used CPU. The observed performance gap between CPU and GPUs is not surprising given the fact that the framework currently performs no implicit cache optimizations like blocking.

The only difference in the definition of single precision and double precision kernels lies in the use of different data types. Generally the performance figures for double precision appear similar to the single precision ones. The GPUs perform slightly better and the CPU shows no significant different behavior. The only exception is the GeMM kernel. Here the CPU performs worse than with single precision because of the increased data size and therefore decreased cache efficiency. On the GPU side the achievable double precision performance is much lower as the single precision performance. This is shown in Table 3. This explains the high efficiency that the GeForce GPU reaches.

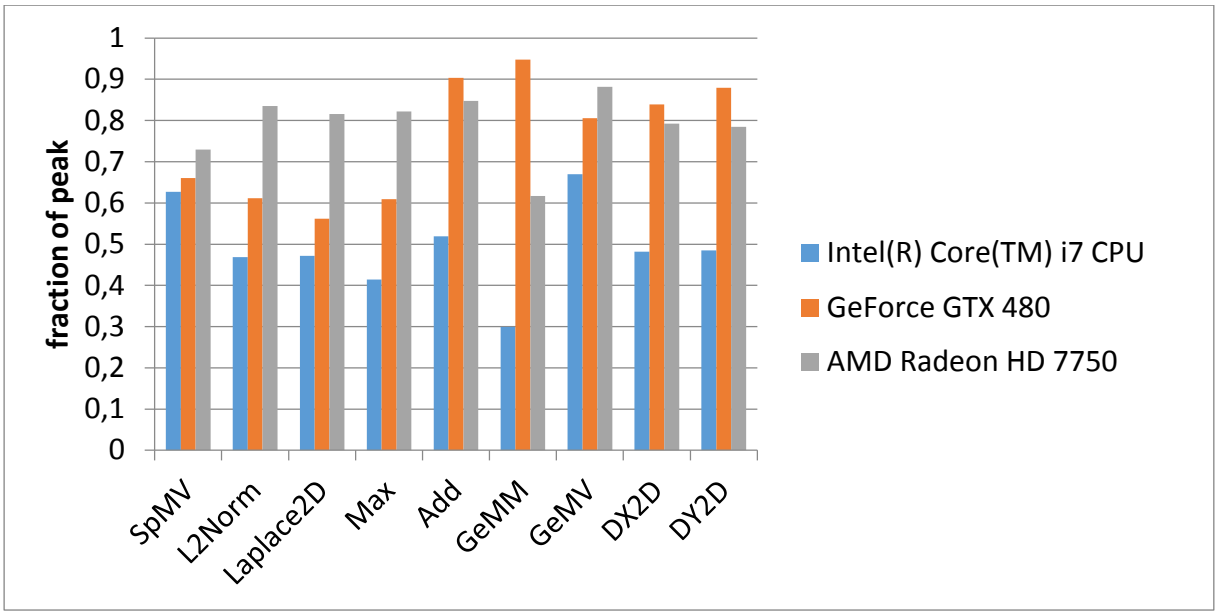

Figure 7: Benchmark of some common double precision kernels that have been implemented in the proposed framework. Shown is the fraction of either the peak performance or the peak bandwidth depending on the limiting factor of the kernel

A more complex example of an algorithm is phase unwrapping. It removes the phase wraps that can appear in interferograms [7]. A straight forward algorithm that solves the problem looks like:

1. Get the derivatives in x- and y- direction of the wrapped phase image
2. Delete jumps in the derivatives that exceed a predefined limit
3. Recalculate the unwrapped phase image from the modified derivatives by integration

The 1st and 2nd step can be implemented straight forward. Step three is more difficult to handle. Because of the modification of the derivatives there is no guarantee that a unique solution for the integration problem exists. Instead of the exact integral, an algorithm that delivers a result with a minimal residual is sought. To solve this problem it is reformulated as a least squares problem. The resulting linear system is solved with a geometric multigrid algorithm.
The complete unwrapping algorithm has been implemented in C++ and parallelized with the help of OpenMP [8]. One run on an Intel® Core i7 960 CPU took 0.25 seconds. The same algorithm implemented with the proposed framework took 0.15 seconds for a run on the same processor and 0.038 seconds on a NVIDIA® GeForce GTX 480 GPU. This leads to the speedup shown in Figure 8.
This speedup made it possible to execute the phase unwrapping algorithm with 26 frames per second and enables its usage in an interferometry live view application. This application is used to measure the wavefront deformation of high performance objectives.

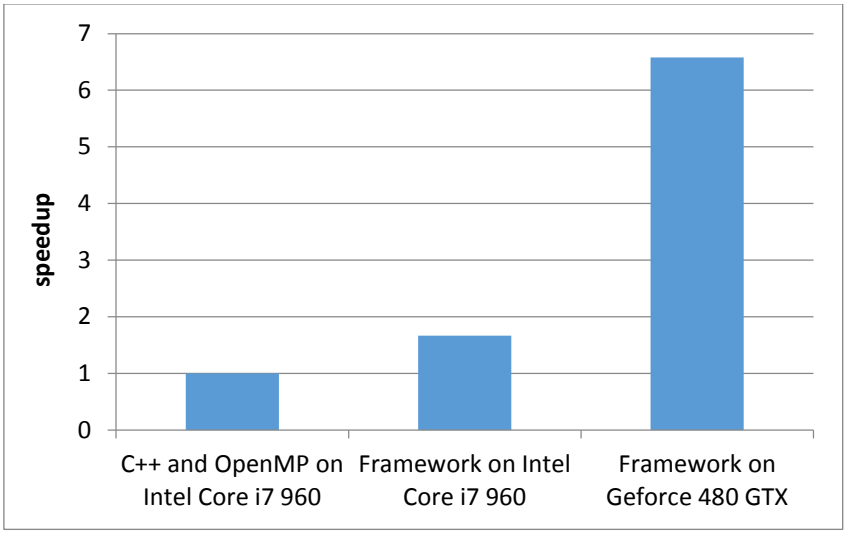

Figure 8: Speedup for the phase unwrapping of a 1024x1024 matrix

5. RELATED WORK

Performance portability has been addressed in many different frameworks and research efforts. One established framework is ATLAS [9]. It focuses on delivering performance portable basic linear algebra subroutines (BLAS). This is mainly accomplished by blocking of memory accesses. The FFTW [10] is a framework that delivers empirical auto-tuned FFT routines. Both mentioned frameworks need complete C99 compiler support and therefore are not useable on architectures like GPUs.

In the last years a number of frameworks have been developed that address the portability of algorithms between completely different architectures like CPUs and GPUs.

ViennaCL [11] is a numerical library that supports BLAS and the iterative solution of sparse systems of equations. The library is based on OpenCL and supports computations on GPUs and CPUs. To increase the performance it auto-tunes the workgroup-dimension of the used kernels [12] and contains a kernel generator that can fuse multiple operations on vectors (BLAS level 1) into one OpenCL kernel [13].

PetaBricks [14] is an implicit parallel language and compiler. Its main paradigm is to describe multiple algorithms for solving one problem and how they fit together. The PetaBricks runtime and compiler not only auto-tunes different parameters like data distribution and accuracy, but also the choice of algorithms.

Halide [15] is an image processing framework that includes an optimizing compiler for the Halide image processing language that is a domain specific language for the description of image processing pipelines. Compiler targets include CUDA, OpenCL and x86/SSE. It uses modelling of the complete processing pipeline to create complex scheduling choices that include different loop unrolling strategies or the use of vector instructions. Exhaustive auto-tuning of the identified scheduling choices creates high efficient code that outperforms even hand tuned solutions.

6. FUTURE IMPROVEMENTS

An obvious challenge for future work is to further improve the reached performance efficiency of the created OpenCL kernels. This could be possible with the introduction of new code variations or the mixed usage of already developed ones.
The benchmarks showed that the reduction algorithms perform mediocre on NVIDIA architectures and not very well on the Intel CPU. This will be addressed. The introduction of cache blocking concepts could also improve the performance on CPU architectures.
A major improvement, that is planned to be implemented, is the support for using multiple heterogeneous devices. But this may need some structural changes in the framework. There is not only the problem of proper load balancing but also the incorporation of boundary handling in the codegenerator and the automated identification of a feasible distribution of the problem to the different devices.
Another aspect, that will be explored, is benchmarking the framework on newer NVIDIA hardware like GeForce® Titan and higher performance AMD GPUs.

7. CONCLUSION

While there is room for future improvements it has been shown that the introduced framework is able to deliver considerable performance on different hardware architectures. This makes it possible to create compute intensive metrology applications that can run on different hardware architectures with high performance without changes in the application code. This performance portability reduces the development time of performance critical software and enables a usage on different hardware architectures. Additionally the ability to use nearly any OpenCL device reduces the dependency on a single hardware vendor or architecture. This can be important for software that should be used for a long time span.

8. ACKNOWLEDGEMENTS

The research in this paper was partly funded by the Thüringer Ministerium für Wirtschaft, Arbeit und Technologie and and the European Social Fund (ESF) through the Thüringer Aufbaubank (TAB) within project no. 2011 SD 0017.

REFERENCES

[1] G. E. Moore, "Cramming more components onto integrated circuits, Reprinted from Electronics, volume 38, number 8, April 19, 1965, pp.114 ff.," *Solid-State Circuits Society Newsletter, IEEE,* vol. 11, no. 5, pp. 33-35, september 2006.

[2] M. J. Flynn, "Some Computer Organizations and Their Effectiveness," *IEEE Trans. Comput.,* vol. 21, no. 9, pp. 948-960, september 1972.

[3] Intel Corporation, "Intel® 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture," february 2014.

[4] J. J. Dongarra, H. W. Meuer, H. D. Simon and E. Strohmaier, "Top500 supercomputer," june 2014. [Online]. Available: http://www.top500.org/lists/2014/06/. [Accessed 1 september 2014].

[5] K. Opencl and A. Munshi, *The OpenCL Specification Version: 1.2 Document Revision: 19,* 2012.

[6] J. Lai and A. Seznec, "Performance Upper Bound Analysis and Optimization of SGEMM

on Fermi and Kepler GPUs," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Washington, DC, USA, 2013.

[7] H. A. Zebker and Y. Lu, "Phase unwrapping algorithms for radar interferometry: residue-cut, least-squares, and synthesis algorithms," *J. Opt. Soc. Am. A,* vol. 15, no. 3, pp. 586-598, march 1998.

[8] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *Computational Science \& Engineering, IEEE,* vol. 5, no. 1, pp. 46-55, 1998.

[9] R. C. a. P. A. a. D. J. Whaley, "Automated empirical optimizations of software and the ATLAS project," *Parallel Computing,* vol. 27, no. 1-2, pp. 3-35, 2001.

[10] M. Frigo and S. G. Johnson, "The Design and Implementation of FFTW3," *Proceedings of the {IEEE},* vol. 93, no. 2, pp. 216-231, 2005.

[11] J. Weinbub, K. Rupp and S. Selberherr, "Towards Distributed Heterogenous High-Performance Computing with ViennaCL," in *LSSC'11: Proceedings of the 8th International Conference on Large-Scale Scientific Computing*, Berlin, Heidelberg, 2012.

[12] P. Tillet, K. Rupp and S. Selberherr, "An Automatic OpenCL Compute Kernel Generator for Basic Linear Algebra Operations," in *HPC '12: Proceedings of the 2012 Symposium on High Performance Computing*, San Diego, CA, USA, 2012.

[13] K. Rupp, J. Weinbub and F. Rudolf, "Automatic Performance Optimization in ViennaCL for GPUs," in *POOSC '10: Proceedings of the 9th Workshop on Parallel/High-Performance Object-Oriented Scientific Computing*, New York, NY, USA, 2010.

[14] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman and S. P. Amarasinghe, "PetaBricks: a language and compiler for algorithmic choice," in *SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[15] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand and S. Amarasinghe, "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines," *SIGPLAN Not.,* vol. 48, no. 6, pp. 519-530, June 2013.

CONTACTS

Dipl.-Inf. Tobias Beier tobias.beier@zeiss.com