# OpenCL, a Viable Solution for High-performance Medical Image Reconstruction?

Christian Siegl[a], H. G. Hofmann[a], B. Keck[a], M. Prümmer[a] and J. Hornegger[a,b]

[a]Pattern Recognition Lab, Friedrich-Alexander-Universität Erlangen-Nürnberg;
[b]Erlangen Graduate School in Advanced Optical Technologies (SAOT), Erlangen, Germany

## ABSTRACT

Reconstruction of 3-D volumetric data from C-arm CT projections is a computationally demanding task. For interventional image reconstruction, hardware optimization is mandatory. Manufacturers of medical equipment use a variety of high-performance computing (HPC) platforms, like FPGAs, graphics cards, or multi-core CPUs. A problem of this diversity is that many different frameworks and (vendor-specific) programming languages are used. Furthermore, it is costly to switch the platform, since the code has to be re-written, verified, and optimized.

OpenCL, a relatively new industry standard for HPC, promises to enable portable code. Its key idea is to abstract hardware in a way that allows an efficient mapping onto real CPUs, GPUs, and other hardware. The code is compiled for the actual target by the device driver.

In this work we investigated the suitability of OpenCL as a tool to write portable code that runs efficiently across different hardware. The problems chosen are back- and forward-projection, the most time-consuming parts of (iterative) reconstruction. We present results on three platforms, a multi-core CPU system and two GPUs, and compare them against manually optimized native implementations.

We found that OpenCL allows to share a common framework in one language across platforms. However, considering differences in the underlying architecture, a hardware-oblivious implementation cannot be expected to deliver maximal performance. By optimizing the OpenCL code for the specific hardware we reached over 90% of native performance for both problems, back- and forward-projection, on all platforms.

**Keywords:** Back-projection, Benchmark, CBCT, CPU, CUDA, CT, Forward-projection, GPGPU, GPU, High-performance, HPC, Multi-core, OpenCL, RECON

## 1. INTRODUCTION

In the field of medical image reconstruction and medical image processing, complex problems often have to be solved with very strict time constraints. Especially in an interventional environment delays are critical. To solve these problems in a reasonable amount of time, high performance computing (HPC) is employed. Today, vendors draw on a variety of HPC solutions. Custom hardware like field-programmable gate arrays (FPGAs)[1] is in use as well as highly optimized CPU implementations,[2] or graphics cards (GPUs).[3,4] While all these platforms offer plenty of parallel computing units, each has its own programming language and tool chain, e.g. VHDL, SIMD intrinsics, OpenGL, or CUDA. To assess the performance of a new platform for an algorithm means to create an implementation optimized specifically for it, using its own language. Updating a product to a new computing platform is considerably impeded, as the new code has to be re-written, tested (FDA approved), and optimized again.

In 2008, a consortium of hardware and software vendors created the Open Computing Language (OpenCL)[5] which aims to address these problems by providing a common framework across platforms. The main goal of OpenCL is to enable a single source code to be executed efficiently on any platform which supports OpenCL. Therefore, it uses a common programming language and a generalized hardware abstraction layer which can be efficiently mapped onto modern heterogeneous parallel architectures like GPUs, CPUs, or the CELL processor.

---

Further author information: Send correspondence to H.G.Hofmann. E-mail: hannes.hofmann@cs.fau.de;
Address: Lehrstuhl fuer Mustererkennung, c/o Hannes G. Hofmann, Martensstr. 3, 91058 Erlangen, Germany.

In this work we answer three obvious questions:

- Is OpenCL code portable, i.e. can a single source code be executed on different platforms?
- Is the shared code efficient (without specific optimizations)?
- Can (optimized) OpenCL code compete with a manually optimized implementation?

The rest of this paper is structured as follows: First, we introduce OpenCL and the algorithms we examine in Sec. 2. Thereafter, in Sec. 3, we describe the hardware used. The various implementations and hardware-specific optimizations are detailed in Sec. 4. Section 5 contains our evaluation methods and the results which are discussed in Sec. 6. Finally, we conclude our findings and present an outlook onto future developments in Sec. 7.

## 2. MATERIALS AND METHODS

Reconstruction of Computed Tomography (CT) data requires vast amounts of computing power. Depending on the reconstruction algorithm, the most computation time is spent on back-projection (analytical algorithms)[6] or back- and forward-projection (iterative methods).[7] Therefore, we chose these problems for our comparison.

### 2.1 Open Computing Language

OpenCL is an open framework for parallel programming of heterogeneous systems. It provides a uniform programming environment for software developers to write efficient and portable code for different platforms. Furthermore, OpenCL C extends C99 by some data types and functions to support parallel processing. The key-concept of OpenCL, however, is a hardware abstraction layer, consisting of a *platform*, *execution* and *memory model*.

The *platform model* consists of a host connected to one or more OpenCL devices. An OpenCL device is divided into one or more compute units (CUs) which are further divided into one or more processing elements (PEs) which are the actual computing units. Execution of an OpenCL program is split into two parts: a host program and kernels which are executed on the host and on one or more OpenCL devices, respectively. The host program defines the context for the kernels and manages their execution.

The OpenCL *execution model* defines how the kernels are executed. When a kernel is submitted for execution by the host program, an index space is defined and an instance of the kernel executed for each point in this index space. One kernel instance is called a work-item, identified by its position in the index space. Each work-item executes the same code, but the specific pathway through the code and the data operated upon can vary per work-item. Work-items are organized into work-groups, providing a more coarse-grained decomposition of the index space.

Using the OpenCL *memory model* the kernel executing work-items have access to four distinct memory regions, each with different restrictions and access speed as shown in Table 1.

### 2.2 Back-projection / RabbitCT

The evaluation of our implementation of back-projection was performed using the open reconstruction benchmark RabbitCT.[8] It comprises a public clinical data set and an open-source test framework that allows for presentation of comparable results. The RabbitCT data set contains already preprocessed projection images. Geometry information is provided using projection matrices. Additionally to performance measurements, RabbitCT allows easy verification of an implementation's correctness.

The data set consists of $N = 496$ projections with a resolution of $S_u = 1248, S_v = 960$ pixels. The size of the reconstructed volume is $512^3$ voxels ($L = 512$).

| **memory type** | global mem. | constant mem. | local mem. | private mem. |
|---|---|---|---|---|
| **accessible by** | host and kernel (r/w) | host (w) and kernel (r) | one work-group (r/w) | per kernel (r/w) |

Table 1. Memory types specified by OpenCL and their accessibility. The order implies the speed, from left (slower) to right (faster).

**Algorithm 1:** Back-projection for the $n$-th projection image.

```
input  : I_n, A_n, f, L
output : f

// for each voxel in the volume f
for x = 0 to L − 1 do
    for y = 0 to L − 1 do
        for z = 0 to L − 1 do
            // update the volume
            f(x, y, z) = f(x, y, z) + 1/w_n(x,y,z)^2 · p̂_n(u_n(x, y, z), v_n(x, y, z));
        end
    end
end
```

With RabbitCT, the part of back-projection that has to be implemented is reduced to incrementing every voxel of the volume with the value from the current projection image. This value is obtained by projecting the voxel onto the projection image using the provided projection matrix.

Since the data is preprocessed, the discrete version of the FDK algorithm[6] breaks down to

$$f(x, y, z) = \sum_{n=1}^{N} \frac{1}{w_n(x, y, z)^2} \cdot \hat{p}_n(u_n(x, y, z), v_n(x, y, z)), \tag{1}$$

where

$$\begin{aligned} u_n(x, y, z) &= (a_0x + a_3y + a_6z + a_9) \cdot w_n(x, y, z)^{-1} \\ v_n(x, y, z) &= (a_1x + a_4y + a_7z + a_{10}) \cdot w_n(x, y, z)^{-1} \\ w_n(x, y, z) &= a_2x + a_5y + a_8z + a_{11} \end{aligned}$$

and

$$\boldsymbol{A}_n = \begin{pmatrix} a_0 & a_3 & a_6 & a_9 \\ a_1 & a_4 & a_7 & a_{10} \\ a_2 & a_5 & a_8 & a_{11} \end{pmatrix} .$$

This particular mathematical notation has been chosen for its closeness to the implementation. $f$ denotes the reconstructed volume, $N$ is the number of projection images, and $\boldsymbol{A}_n$ is the projection matrix. The function $\hat{p}_n : \mathbb{R} \times \mathbb{R} \mapsto \mathbb{R}$ performs a bilinear interpolation with a zero boundary condition in the projection image $\boldsymbol{I}_n$. Pseudo code of the back-projection is given in Algorithm 1. For more details we refer the reader to the original RabbitCT article.[8]

### 2.3 Forward-projection

We use a volumetric ray casting approach for the forward-projection step. Its basic functionality has been described in literature.[4,9] The attenuation observed by every detector pixel is determined for every view. Therefore, a ray $\vec{r}$ is drawn pointing from the X-ray source position $\vec{s}_n$ towards the pixel position $(u, v)$. Then, the densities $\rho$ inside the volume are sampled equidistantly along the ray using trilinear interpolation $\hat{f}$. The accumulation of these sampling values is the attenuation value in the projection, approximating Beer's law (cf. Eq. (2)).

$$I = I^0 \cdot exp\left(-\int \rho(\vec{r}(t))\, dt\right) \tag{2}$$

Given a proper pre-processing of the projection images, which are corrected for the source intensity $I^0$, only the integral has to be computed in the forward projection step. Subtraction of this forward-projected volume from

**Algorithm 2:** Forward-projection for the $n$-th projection image.

```
input  : I_n, A_n, f, S_u, S_v
output: D_n
calculate s_n, using A_n
for v = 0 to S_v do
    for u = 0 to S_u do
        calculate r, the ray from s_n to (u, v), using A_n
        for each sample point x along r do
            ProjectedValue = ProjectedValue + f^(x)
        end
        D_n(u, v) = I_n(u, v) − ProjectedValue
    end
end
```

the $n$-th original projection image $\boldsymbol{I}_n$ yields the error $\boldsymbol{D}_n$ of the current volume estimate w.r.t. $\boldsymbol{I}_n$ (cf. Eq. (3)). Pseudo code of the forward-projection is shown in Algorithm 2.

$$\boldsymbol{D}_n(u,v) = \left(\boldsymbol{I}_n(u,v) - \sum_t \hat{f}(\vec{sp}_n(t,u,v))\right), \tag{3}$$

where the sampling points $\vec{sp}$ are calculated as

$$\vec{sp}_n(t,u,v) \quad = \quad \vec{s}_n + t \cdot \alpha \cdot \vec{r'}_n(u,v).$$

$\vec{r'}_n(u,v)$ denotes the normalized directional vector of the current ray $\vec{r}$, and $\alpha$ the sampling step size.

To compensate for the error of the current volume estimate, $\boldsymbol{D}_n$ is back-projected into the volume after applying a relaxation factor $c$.

# 3. HARDWARE AND OPENCL-FRAMEWORKS

We ran our tests on three different HPC systems:

- NVIDIA Tesla C1060; 4 GB device memory; CUDA 3.1 driver, including OpenCL 1.1.
- ATI Radeon HD 5870; 1 GB; ATI Stream SDK 2.3 including OpenCL 1.1.
- Intel Core2 Extreme X9650 CPU at 3.0 GHz; 4 GB RAM; ATI Stream SDK 2.1 including OpenCL 1.0.

At the time of writing ATI's Stream SDK was the only framework featuring support for x86 CPUs.

# 4. OPENCL-SPECIFIC IMPLEMENTATION DETAILS

## 4.1 Back-projection

For one view, all voxels can be back-projected independently from each other. Therefore, parallelizing over the voxels is the most efficient way. Although it has some potential to break the outer loop (over all projections) we kept that one intact throughout all implementations. After all, we are not looking for the fastest back-projection implementation, but for an assessment of OpenCL, and a fair comparison across hardware platforms.

**Generic OpenCL Implementation:** The first OpenCL implementation was similar to the LolaBunny reference implementation from RabbitCT (cf. Algorithm 1). To exploit OpenCL's support for 3-D problems, we eliminated the inner loops ($x$, $y$, and $z$) in the kernel. Consequently, there was one kernel instance per voxel. This implementation did not use OpenCL vector data types like float4.

**Optimizations for Multi-core CPU:** The generic version creates $512^3$ work-items (per projection image), introducing lots of overhead. Therefore, the problem was re-organized as a 2-D problem to better suit the CPU's

relatively low number of processing elements. Further, to enable efficient loading and storing of the volume data, the innermost loop was transformed such that it traverses the volume in $x$-direction. Then, voxels processed in consecutive iterations of the innermost loop are adjacent in memory, exploiting the concept of cachelines used by CPUs to improve memory latencies.

The second optimization step was to exploit the vector units by using OpenCL's vector data types. The OpenCL-C language used for kernel code supports arithmetic operations with vector data types which simplifies vectorization and increases readability.

**Optimizations for NVIDIA GPU:** GPUs provide many simple processing elements with plenty of computing power. Furthermore, they feature special hardware units to make context switches very fast. So having a lot more work-items than processing elements is desirable to hide memory latency. The generic implementation's partitioning as a 3-D problem ensured that many threads were available (NV1).

The first GPU-specific optimization was the use of OpenCL's `image2d_t` type (NV2). This image data type not only provides the use of the graphics card's texture caching capabilities but also allows utilization of hardware texture interpolation. Note that `image2d_t` is not fully supported by all OpenCL frameworks, yet.

In addition to many kernel instances, to get good throughput, memory accesses on graphics cards should meet some constraints. Maximum performance is achieved when work-items within one work-group access contiguous memory locations. This is called *coalesced memory access*.
To get better control over memory accesses and achieve higher performance, the problem was reorganized as a 2-D problem, where every kernel loops over all voxels in $y$-direction, analogous to the CUDA optimizations in Scherl *et al.*[3] (NV3). Together with work-groups that process all voxels in one line in $x$-direction, coalesced memory access is ensured.

**Optimizations for ATI GPU:** Considering that the code again was targeted for a GPU platform we did not expect too many changes to the NVIDIA code.

However, despite the device memory of 1 GB, the driver of the ATI GPU did not let us allocate buffers larger than 128 MB (and 512 MB total). Only after setting two environment variables, we were able to allocate buffers of 256 MB (1 GB total). This required splitting of the volume into two regions and some adaptions inside the kernel. Therefore, the NVIDIA version was adapted to these constraints (ATI1).

As the performance of the adapted NVIDIA code did not meet our expectations, different problem organizations were investigated. Since the technical documents by ATI don't emphasize coalesced memory access as much as NVIDIA's documents, our second approach was to increase the number of work-items and again use a 3-D problem. The idea behind this is to increase the number of work-items and thus improve the possibility of hiding memory latencies. Two versions of this approach were tested, one with one voxel per kernel (ATI2) and one with two voxels per kernel (ATI3) originating from the two memory regions.

The fourth implementation (ATI4) used vectorized code, similar to the CPU-optimized version, but with a memory access pattern suitable for graphics cards.

## 4.2 Forward-projection

All pixels can be forward-projected independently from each other. Therefore, parallelizing the forward-projection over the pixels is the most efficient way.

**Optimizations for Multi-core CPU:** As runtimes of the proposed algorithm for iterative reconstruction are far beyond practical relevance the performance of iterative reconstruction on the CPU was not investigated. On a CPU one would rather use a simpler algorithm for forward-projecting rays, e.g. Joseph's method.[10] Comparing against this would make no sense in this paper.

**Optimizations for NVIDIA GPU:** Every kernel processes one ray and sums up all volume elements along it, including trilinear interpolation. As learnt from the back-projection, expoliting the hardware texture interpolation and caching is mandatory. Therefore, the volume data is stored as an `image3d_t`. Also, the difference image is stored as a texture, such that it can be used as input to the consecutive back-projection step without copying.

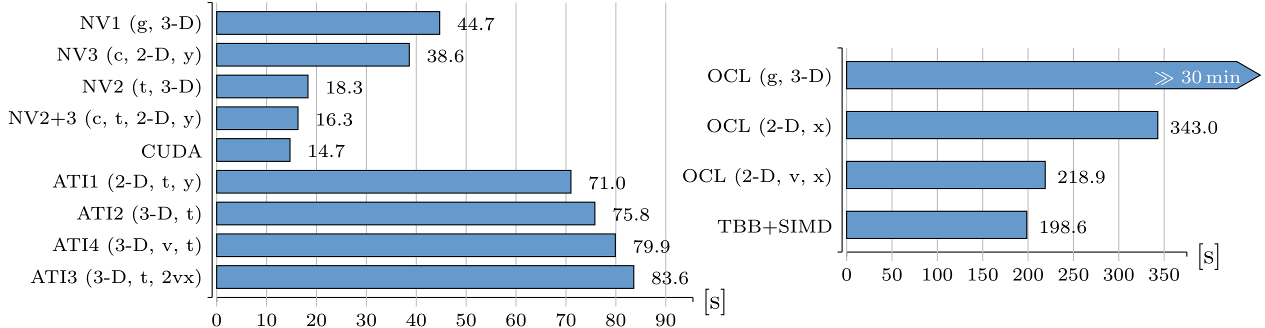

Figure 1. Back-projection runtimes on GPU (left) and CPU (right) in [s]. Shorter bars are better. In the left figure, the top bars are for the NVIDIA card (NV), the bottom ones for ATI. (g) means generic, (c) coalesced, (t) textures units, (v) vectorized. (x) and (y) denote the index of the innermost loop, (2vx) denotes the kernel that processes two voxels.

Note that the volume cannot be a 3-D texture in the back-projection since it is read and written to, which is not allowed within one kernel. One possible solution would be to copy the volume to the host at the end of the back-projection and load it back as a 3-D texture. However, the low speed of the PCIe-bus is prohibitive. Or, one can hold two copies of the data on the GPU. This doubles the memory requirement, but updates are very fast (see also Keck *et al.*[4]).

**Optimizations for ATI GPU:** As described above, an efficient implementation requires two volume copies on the device. Since the ATI GPU used to test performance on only was equipped with 1 GB of graphics memory, we decided to run only half of the problem ($512 \times 512 \times 256$) when making comparisons with this GPU.

Copying the volume within the device memory was more than 10 times slower on our ATI GPU with the current ATI framework than with the NVIDIA card. Therefore, another solution for the synchronization of the two memory regions had to be found. In contrast to current NVIDIA GPUs, the ATI GPU features the possibility to write to `image3d_t`-textures from a kernel. Still, two volume copies are required, since read and write from within one kernel is forbidden.

However, with 3-D texture writes, we can get rid of the slow copy operation. The back-projector stores the updated voxels not only to its `float` memory buffer, but also to the forward-projector's `image3d_t` texture. This additional write operation had no impact on the runtime of the back-projecting kernel and thus copying the volume into the texture is for free.

## 5. EVALUATION AND RESULTS

All benchmarks of the back-projection were run with RabbitCT. We also extended the open-source framework to support iterative reconstruction.

The data set consists of 496 projections with a resolution of $1248 \times 960$ pixels. The size of the reconstructed volume was $512^3$ voxels.

### 5.1 Back-projection

The CUDA reference implementation was SpeedyGonzales, as published on www.rabbitct.com. The CPU reference implementation was manually optimized as described in Hofmann *et al.*[2] (TBB+SIMD). The runtimes of all our implementations on all three systems are shown in Figure 1.

The right diagram of Figure 1 presents the runtimes on the CPU system. The generic version took longer than 30 minutes, hence we do not give an exact number here. Using a 2-D partitioning reduces the runtime to $1.7\times$ the optimized version. Finally, the version using vector types achieved 91% of the native implementation's performance.

The top bars in the left diagram of Figure 1 show that the runtime of the generic version on the NVIDIA GPU is about $3\times$ slower than the CUDA version. Applying optimizations (using texture samplers and coalesced

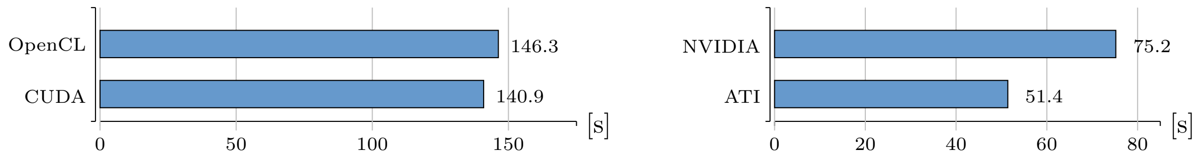


Figure 2. Forward-projection runtimes in [s]. Left: NVIDIA GPU, OpenCL vs. native. Right: NVIDIA and ATI, both OpenCL, only half volume. Shorter bars are better.

memory access) improved performance. Finally, a version with both optimizations was able to achieve 90% of the CUDA implementation's performance.

The bottom bars in Figure 1 (left) show the performance on the ATI GPU. Despite some efforts in optimizing the implementation, the runtime of the different variants does not vary much. Even the fastest implementation on ATI's graphics card is 4.4× slower than what was achievable on the NVIDIA GPU.

### 5.2 Forward-projection

Figure 2 (left) shows that, only 4% behind, the OpenCL forward projection was almost as fast as the native CUDA implementation on the NVIDIA GPU.

The right diagram in Figure 2 shows that for the forward-projection task the ATI Radeon HD 5870 is able to outperform NVIDIA's Tesla C1060 with only 68.4% of the runtime. As mentioned in Sec. 4 the runtimes provided for the forward-projection are for half the problem due to the limited memory size of our ATI graphics card.

## 6. DISCUSSION

During this work it became obvious that OpenCL allows to have a single codebase for all platforms which support it. However, one cannot expect to get one solution which is efficient on all platforms. This is due to several reasons:

- Fundamental differences in memory layout, size, and latency (e.g. device memory on PCIe boards)
- Fundamental differences in platform layout (e.g. number of cores between 1 and several hundreds)
- Fundamental differences in execution model (OS threads vs. hardware threads)
- Hardware-specific features (e.g. texture caching and samplers on GPUs)

Extensions supporting specific hardware features allow optimization, but break portability. While OpenCL offers no benefit in terms of performance, yet, the code is no longer vendor-specific and should also be executable on other devices which support OpenCL.

The performance results on the benchmarked systems are almost on par with the native implementations. The small penalty of about 10% is most likely attributed to the compilers which are not as sophisticated, yet. We expect this gap to become smaller over time.

On the CPU, vector data types keep the vectorized code readable and very close to the scalar version. The effort for optimization is reduced to a minimum. The smaller than expected performance gain from vectorizing the code can easily be explained by the fact that the compiler already does a good job in vectorizing the non-vectorized version. The runtimes show that this automatic vectorization at the current point in time can't keep up with a hand vectorized version of the code.
The used version of the ATI Stream SDK does not support image data types on CPU devices. Hence, code that uses this feature is not portable. Instead, bilinear interpolation had to be implemented manually in the CPU version.
Another inconveniency stems from OpenCL's memory model: even on a CPU host and device memory are separate. Consequently, data is copied by the driver and memory consumption rises. This can be avoided by using *host pointers*, but they would degrade performance on the GPU.

On NVIDIA GPUs, OpenCL and CUDA offer very similar concepts and even share intermediate binary formats. Both are sensitive to hardware-specific optimizations such as the size of the work-groups or coalesced

memory access. In order to achieve maximum performance, GPU-specific features must be exploited. The runtime differences between OpenCL and CUDA can be explained by the maturity of the compilers. NVIDIA released their CUDA SDK before OpenCL became available and the compiler for CUDA thus is more optimized and evolved. It can be expected that this gap closes over time and there will be no more or only marginal differences in future versions.

On the ATI GPU we saw mixed results. Looking at the theoretical peak performance, the card should easily be able to outperform every other system in our test setup. In the evaluation, however, it could not meet up to this expectation. In reality, only for the forward-projector the performance was better, but drastically worse for the back-projector, than on the NVIDIA counterpart. From running the code in different configurations and review of technical documents we conclude that this is due to the following factors.
Compared to NVIDIA's cards, current ATI graphics cards feature a main difference in device architecture. On ATI GPUs the stream cores are subdivided into five processing elements (not correlated to OpenCL processing elements). While this division dramatically increases the theoretical peak performance of the card, it is more or less transparent to the programmer and only accessed by the compiler. In our experiments the so called packing of the code onto these five processing elements never utilized more than 60% of the GPU (max. 38% for the back-projector).
The used ATI graphics card has a compute unit to texture unit ratio of 4 to 1 compared to a ratio of 3 to 1 on the NVIDIA card. That means more processing elements have to share one texture unit and thus memory latency goes up.
The last factor that impacts performance on the ATI graphics card is global memory access. It can be seen that the forward-projection kernel, which only works on image data types, performs quite well, while the back-projection kernel, which heavily uses global memory, performs much worse than expected. We ran this kernel in many different configurations (e.g. only memory access, or no memory access, etc.) and we can conclude that the limiting factor for performance is global memory access.

One key point in optimizing performance on all platforms that has very high impact on runtimes is the size of work-groups in different dimensions. The layout of a work-group specifies how many points in the kernel index space are processed by the work-group. Things to keep in mind when determining the optimal layout are, for example, locality to optimize usage of the texture cache (resulting in a more cubic layout), or coalescing of memory access (a layout that stretches across one line in index space). The work-group sizes have to be chosen in a way that the global problem size in every dimension is divisible by the respective dimension of the work-group size and the product of the work-group size in all dimensions should be a multiple of the smallest possible execution unit on the used device. When it comes to work-group size it is important to know that some experiments have to be done as devices often behave very counterintuitive. The presented results always use the fastest configuration that has been found in experiments.

To set these problems into perspective it has to be considered that differences in source code, even between optimized implementations, are small. Some of these changes could further vanish if vendors would emulate functionality not provided by certain devices. For example, if `image2d_t` and sampling would be handled by the OpenCL framework/driver on the CPU, portability would increase a lot. For similar architectures, like different graphics cards, similar optimizations are efficient. For example, ATI1, the implementation closest to the fastest NVIDIA code, was also the one performing best on ATI. And even despite the fact that code has to be optimized for different platforms independently it is important to keep in mind that all this happens in one programming language using the same framework. This heavily reduces the complexity of this task.

## 7. CONCLUSION AND OUTLOOK

Picking up the questions that motivated this work we can conclude that it is well possible to write platform independent software with OpenCL. Shared code, once optimized for a specific architecture, may also perform well on another one (e.g. NV2+3 and ATI1). However, if the architectures are too different, specific optimizations will be necessary. Optimization is eased by OpenCL features, the fact that a common language is used, and the fact that only minor changes are required. By optimizing the implementations for the different platforms we showed that OpenCL can perform on a level close to highly optimized native implementations. Since OpenCL

is supported by many vendors we expect it to become even better rapidly and compilers to perform many of the optimizations automatically.

Two recent facts support these statements: The latest version of the ATI Stream SDK that was released shortly after the evaluation for this paper was done supports image data types on the CPU. This means portability increases. Another fact is the recent release of Intel's OpenCL framework. The framework is still in alpha stage but proves that all major vendors invest into OpenCL.

In summary, we conclude that interoperability is better than with any other framework we know of and we encourage programmers to start using OpenCL.

## REFERENCES


[1] Heigl, B. and Kowarschik, M., "High-speed reconstruction for C-arm computed tomography," in [*9th International Meeting on Fully Three-Dimensional Image Reconstruction in Radiology and Nuclear Medicine*], 25–28, www.fully3d.org, Lindau (July 2007).

[2] Hofmann, H. G., Keck, B., Rohkohl, C., and Hornegger, J., "Putting 'p' in RabbitCT – Fast CT reconstruction using a standardized benchmark," in [*PARS-Mitteilungen (ISSN 0177-0454)*], 91–100, Gesellschaft für Informatik e.V., Bonn (Dec. 2009).

[3] Scherl, H., Keck, B., Kowarschik, M., and Hornegger, J., "Fast GPU-Based CT Reconstruction using the Common Unified Device Architecture (CUDA)," in [*Nuclear Science Symposium Conference Record, 2007. NSS '07. IEEE*], Frey, E. C., ed., **6**, 4464–4466 (Oct. 2007).

[4] Keck, B., Hofmann, H., Scherl, H., Kowarschik, M., and Hornegger, J., "GPU-accelerated SART reconstruction using the CUDA programming environment," in [*SPIE Medical Imaging Conference Proc.*], Samei, E. and Hsieh, J., eds., **7258**, 72582B.1–72582B.9 (2009).

[5] Khronos Group, "OpenCL 1.1 Specification." Website (2010). Available online at http://www.khronos.org/opencl/.

[6] Feldkamp, L., Davis, L., and Kress, J., "Practical Cone-Beam Algorithm," *Journal of the Optical Society of America* **A1**(6), 612–619 (1984).

[7] Andersen, A. and Kak, A., "Simultaneous Algebraic Reconstruction Technique (SART): A superior implementation of the ART algorithm," *Ultrasonic Imaging* **6**, 81–94 (Jan. 1984).

[8] Rohkohl, C., Keck, B., Hofmann, H. G., and Hornegger, J., "RabbitCT—An Open Platform for Benchmarking 3-D Cone-beam Reconstruction Algorithms," *Medical Physics* **36**, 3940–3944 (Sept. 2009).

[9] Engel, K., Hadwiger, M., Kniss, J. M., Rezk-Salama, C., and Weiskopf, D., [*Real-time volume graphics*], AK Peters (2006).

[10] Joseph, P. M., "An improved algorithm for reprojecting rays through pixel images," *IEEE Transactions on Medical Imaging* **MI-1**(3), 192–196 (1982).