An OpenCL-based Parallel Coder for Erasure-tolerant Storage

Peter Hegen
Institute of Computer Engineering
University of Luebeck, Germany

Peter Sobe
Faculty of Mathematics and Computer Engineering
Dresden University of Applied Sciences, Germany
sobe@htw-dresden.de

Abstract: Erasure-tolerant coding protects against data loss (erasure) in storage systems by calculating redundant data elements and spreading original and redundant data across several devices. This allows to reconstruct data in case of device failures. Code calculations can be done on the GPU using OpenCL. This allows a highly parallel execution and moves the computational cost away from the CPU. In this paper we introduce a methodology for coding using OpenCL. It consists in a preparation of XOR equations and their compilation to OpenCL kernels.

1 Introduction

Erasure-tolerant coding is required to protect large data memories and long term storage against partial data loss due to failures. This is important because storage and memories are often build from many devices in order to allow parallel access and to aggregate their capacities. However, accessing a number of devices in parallel decreases the systems reliability. Erasure-tolerant codes counteract against this effect and allow to tolerate failures. In case of failures, lost data elements are recalculated from remaining data elements using additional data at the redundant elements. Erasure-tolerant coding can be placed in different system levels, ranging from the application processes, over middleware and operating system components to pure hardware architectures.
A variety of different codes exist. Some of them are capable to detect data errors only, some other to correct them and a certain class of codes solely tolerate loss of data on places that are a priori known. The latter ones are called erasure-tolerant codes and are sufficient, when storage devices are diagnosed as faulty, e.g. sectors are not read due to an error that is detected by a checksum test. Most of the codes use a very simple operation for en- and decoding - the XOR operation. For encoding, several XOR sums are calculated over a certain selection of data blocks each. It depends on the specific code and its parameters which blocks are selected

to be XORed. The same principle is applied for decoding - erased data blocks are calculated by XOR sums over selected blocks of the available original data and the redundant data. The block selection for XORing is determined by the number of failures that have to be tolerated and the specific failure case.
In our storage system we moved the part of forming the appropriate selection away from the en- and decoder, by presenting XOR equations for en- and decoding. In the paper we describe how the equations are mapped to OpenCL programs that run on multi core processors and GPUs.
The contribution of the paper is a method for parallel coding on multi core platforms and GPUs that are programmed in OpenCL specifically to the encoding and decoding calculations. We distribute the workload according to a combination of functional parallelism and data parallelism.
The paper is organized as follows. Related work is surveyed in Section 2. The principle of equation-oriented en- end decoding is explained in Section 3 and in Section 4 we describe an implementation using OpenCL. A performance evaluation of our implementation can be found in Section 5. We conclude with a summary.

2 Preliminaries and Related Work

Parallel storage devices and coding first have been introduced with RAID systems [KGP89] in the context of several host-attached magnetic disks and got later adopted to networked storage. Besides simple parity codes that tolerate a single fault, codes for two and more failed devices were found, e.g. the Evenodd code [BBBM95]. Generally, codes base on a distribution of original data across k devices and a number of redundant data blocks that are placed on m additional devices. Well designed codes allow to tolerate every combination of up to m failed devices among these $k+m$ devices in total. Most research has been directed to find codes that show this optimal property for a large range of parameters k and m. Another issue is the number of operations for encoding and decoding that should be as low as possible.
The Reed/Solomon code [IR60] (R/S) is a very general code that allows to construct systems with any k and m. R/S follows a linear equation system approach and requires Galois Field arithmetics. Therefore, R/S needs more instructions and processing time on general purpose CPUs, compared to XOR based codes that directly use the processors XOR instruction. An XOR-based variant of R/S was found by Blomer et al. [BKK$^+$95] and got later known as the so called Cauchy-Reed/Solomon code (CRS). This code divides each of the $k+m$ storage resources into ω different units (ω is chosen such that $2^\omega > k + m$ holds) that are individually referenced by XOR-based calculations. Other codes do exist [PLS$^+$09] but CRS provides for the flexibility needed in this context. A wide variety of XOR-based codes are implemented in software-based system and software libraries, e.g. jerasure[Pla07] that implements general matrix-based codes with optimizations for en- and decoding performance. In our previous work on the NetRAID [Sob03, SP06]

system an equation-based description of encoding and decoding was developed and allows a flexible use of different codes. The coding software was initially a sequentially operating part of the system. Later on, the system allowed to specify coding modules, e.g. for including hardware accelerators [SH07].
XOR-based codes such as CRS are often computed using general purpose CPUs which are able to perform XOR-operations with a width of typically 64 Bit. We could measure a pure XOR throughput of 8 GByte/s on a 2.2 GHz Opteron CPU [SH07]. This result means that under ideal conditions data of size 8 GByte could be XORed with another 8 GByte of data within a second. Unfortunately, CRS coding only reaches a small fraction of this rate. One reason for that are memory accesses that slow down the computation rate. In addition, data parts must be XORed several times, which decreases the coding throughput. Thus it is necessary to exploit further parallelism for coding to reach reasonable high coding throughput that keeps track with the speed of the storage system. It is obvious to use multi core processors. In addition, R/S and CRS have been offloaded to FPGA [HKS+02],[HSM08], GPU using NVIDIA CUDA [CSWB08] and other hardware [SPB10]. These approaches focused on using a single type of accelerator to speed up computations. In [BE09] a Linux block device is combined with a GPU for R/S coding and provides considerable coding speedup compared to a CPU.

3 Coding by Equations

The concept to describe encoding and decoding by XOR equations has been introduced in [SP08]. Equation sets are given as input to the encoder and decoder that solely acts as an engine to do the calculations according to the equations given. The equations are provided by a tool that includes all the CRS arithmetics and provides the equations.
The naming scheme for the units and the placement of the units on the storage resources is as follows. We place units $0, 1, \ldots \omega$-1 consecutively on the first original storage resource, units ω to $2 \cdot \omega$-1 on the second storage resource and so on. Each unit is addressed by the character 'u' and the number, e.g. $u0$ for the first unit in the system. The code calculations reference these units properly by XOR equations. For the 5+2 example, 6 equations are provided - for the 6 units placed on 2 redundant storage resources (see Listing 1).
The particular code, the execution time and resource requirements of coding proce-

```
u15 = XOR(u2,u3,u4,u5,u7,u9,u11,u12)
u16 = XOR(u0,u2,u3,u7,u8,u9,u10,u11,u13)
u17 = XOR(u1,u3,u4,u6,u8,u10,u11,u14)
u18 = XOR(u0,u2,u4,u6,u7,u8,u11,u12,u13)
u19 = XOR(u0,u1,u2,u4,u5,u6,u9,u11,u14)
u20 = XOR(u1,u2,u3,u5,u6,u7,u10,u12)
```

Listing 1: Example for a code (direct 5+2, $\omega = 3$) provided by XOR equations.

dures depend on the equations. The equations in Listing 1 allow to calculate every redundant unit independently from the other ones. Such a coding naively supports parallel processing, but contains redundant calculations, e.g. $XOR(u2, u3)$ is calculated 3 times. We call this the *direct coding style*. In another coding style, the elements are calculated serially, using previously calculated elements when possible. Then redundant calculations can be eliminated, e.g. $XOR(u2, u3)$ is stored in a temporary unit $t0$ and then referenced 3 times. Replacing all common subexpressions by temporary units reduces significantly the number of XOR operations. For the 5+2 system a reduction from 45 to 33 XOR operations occurred. This coding style is called the *iterative style*. For the 5+2 example, the equations are given in Listing 2 with temporary units denoted by 't' and their number. The iterative equations can be formed from the equations given in the direct style using an automated preprocessing step.

```
u15 = XOR(t1,t3,t4)              t0 = XOR(u2,u3)      t6 = XOR(u10,u11)
u16 = XOR(t4,t6,t7)              t1 = XOR(u4,u5)      t7 = XOR(u13,t5)
u17 = XOR(u3,u4,u8,t6,t9)        t2 = XOR(u7,u9)      t8 = XOR(u1,u6)
u18 = XOR(u2,u4,u6,u7,t3,t7)     t3 = XOR(u11,u12)    t9 = XOR(u14,t8)
u19 = XOR(u0,u2,u9,u11,t1,t9)    t4 = XOR(t0,t2)
u20 = XOR(u5,u7,u10,u12,t0,t8)   t5 = XOR(u0,u8)
```

Listing 2: Code (5+2, $\omega = 3$) in an iterative style.

4 Parallel Coding using OpenCL

In this section we explain how the OpenCL code is generated from the equations and how equations are distributed across several kernels for different devices.
The Open Computing Language (OpenCL) is a standard for parallel programming across CPUs, GPUs and other processors [ope09]. OpenCL executes so called kernel functions on devices that operate in a data parallel style. An example of a kernel is given in Listing 3 for element wise multiplications within two arrays. Kernels can be run on the CPU or on accelerators, like GPUs [Khr09]. In addition, operations can be distributed on several devices. In order to support hardware unknown at the time of program development, OpenCL kernels have to be compiled at runtime for each device [ope09]. Each platform is therefore delivered with a compiler that translates the OpenCL kernel to an architecture-specific binary. We use one kernel function for data parallel execution on a device. If many devices are present, equation-oriented parallelism is added.
Compilation at runtime introduces overhead for starting the coding function, but also generates a binary that specifically implements the code. This means that every equation is directly represented in the binary code. In a common approach, control structures (if-then-else, do-while etc.) would be applied to select the appropriate units for XOR-ing and for going through the equations as specified by the parameters. Without control structures, the execution is disburdened from over-

```
__kernel void arraymul(__global const float *a, __global const float *b,
    __global float c)
{
  int i = get_globel_id(0);
  c[i]= a[i] * b[i];
}
```

Listing 3: Example of a OpenCL kernel.

head. An example is shown in Listing 2 where u0 through u14 reference the units for data storage. These code equations are then transformed into the OpenCL code shown in listing 4 with n and r being pointers to memory, to which the data supplied by the program and the redundancy information computed by the OpenCL kernel are written respectively. The offsets are based on the strip size used and are needed to reference the correct bytes in memory.
An advantage is that the kernel may adapt better to the hardware on which it is to

```
__kernel void nr_encode16 (__global char16* n, __global char16* r)
{
 unsigned int i = get_global_id(0);
 char16 t0 = n[131052+i]^n[196578+i];
 char16 t1 = n[262104+i]^n[327630+i];
 char16 t2 = n[458682+i]^n[589734+i];
 char16 t3 = n[720786+i]^n[786312+i];
 char16 t4 = t0^t2;
 char16 t5 = n[0+i]^n[524208+i];
 char16 t6 = n[655260+i]^n[720786+i];
 char16 t7 = n[851838+i]^t5;
 char16 t8 = n[65526+i]^n[393156+i];
 char16 t9 = n[917364+i]^t8;
 r[0+i] = t1^t3^t4;
 r[65526+i] = t4^t6^t7;
 r[131052+i] = n[196578+i]^n[262104+i]^n[524208+i]^t6^t9;
 r[196578+i] = n[131052+i]^n[262104+i]^n[393156+i]^n[458682+i]^t3^t7;
 r[262104+i] = n[0+i]^n[131052+i]^n[589734+i]^n[720786+i]^t1^t9;
 r[327630+i] = n[327630+i]^n[458682+i]^n[655260+i]^n[786312+i]^t0^t8;
}
```

Listing 4: OpenCL kernel for generated from listing 2.

run. For instance, preferred vector sizes may be used to yield greater performance. The kernel in listing 4 was generated for an Intel Q6600 CPU using the OpenCL x86 driver from AMD/ATI. In this case 16 Bytes is the preferred vector length yielding better performance while another kernel version is still needed to encode the end of the file which may not have a size that is a multiple of 16 Bytes. Both – kernels for decoding and encoding – have in common that they are generated according to a distribution scheme. Kernels are generated for each platform separately. In case multiple devices are present, each device computes different types of information (for example devices might calculate redundancy information for only one redundancy storage resource each) depending on the distribution algorithm.

Listings 5 and 6 show an example where each kernel only calculates approximately half the redundancy information and is generated for a different device to be executed on.

```
__kernel __attribute__((vec_type_hint(char16)))
void nr_encode16 (__global char16* n, __global char16* r)
{
   unsigned int i = get_global_id(0);
   r[0+i] = n[3000+i] ^ n[4500+i] ^ n[6000+i] ^ n[7500+i] ^ n[10500+i] ^ n
       [13500+i] ^ n[16500+i] ^ n[18000+i];
   r[1500+i] = n[0+i] ^ n[3000+i] ^ n[4500+i] ^ n[10500+i] ^ n[12000+i] ^
       n[13500+i] ^ n[15000+i] ^ n[16500+i] ^ n[19500+i];
   r[3000+i] = n[1500+i] ^ n[4500+i] ^ n[6000+i] ^ n[9000+i] ^ n[12000+i]
       ^ n[15000+i] ^ n[16500+i] ^ n[21000+i];
}
```

Listing 5: OpenCL kernel generated for a multiple device scenario, computing only half the redundancy information for a $5 + 2$ encoding scheme.

```
__kernel __attribute__((vec_type_hint(char16)))
void nr_encode16 (__global char16* n, __global char16* r)
{
   unsigned int i = get_global_id(0);
   r[4500+i] = n[0+i] ^ n[3000+i] ^ n[6000+i] ^ n[9000+i] ^ n[10500+i] ^ n
       [12000+i] ^ n[16500+i] ^ n[18000+i] ^ n[19500+i];
   r[6000+i] = n[0+i] ^ n[1500+i] ^ n[3000+i] ^ n[6000+i] ^ n[7500+i] ^ n
       [9000+i] ^ n[13500+i] ^ n[16500+i] ^ n[21000+i];
   r[7500+i] = n[1500+i] ^ n[3000+i] ^ n[4500+i] ^ n[7500+i] ^ n[9000+i] ^
        n[10500+i] ^ n[15000+i] ^ n[18000+i];
}
```

Listing 6: OpenCL kernel for the other half of redundant data, related to listing 5.

Kernels for encoding are generated once during program initialization. Especially for encoding, pre-compiled kernels are applicable. Even, if storage units fail during encoding the coding does not change and the kernels can be executed as in a fault free scenario. Data that would be placed on storage devices that failed can simply be discarded or redistributed to other devices.
Contrary to the encoding case, **kernels for decoding** have to be prepared specifically for the failure situation. Kernels are therefore generated once the actual failure locations are known to the decoder.

Multiple Device Load distribution: An equation-based load distribution was chosen to address the existence of multiple OpenCL devices, e.g. a GPU together with a multi core CPU. The equations are grouped into several smaller sets which are then assigned to different devices. Figure 1 illustrates a $5+2$ ($\omega = 3$) case where two equations each are assigned to different devices. When the iterative coding style is applied, only equations that produce a real information unit are distributed. Those equations for the temporary units (expressing common subexpressions) are

executed on all devices that need these units. The load distribution is based on initial profiling and does not change during runtime.

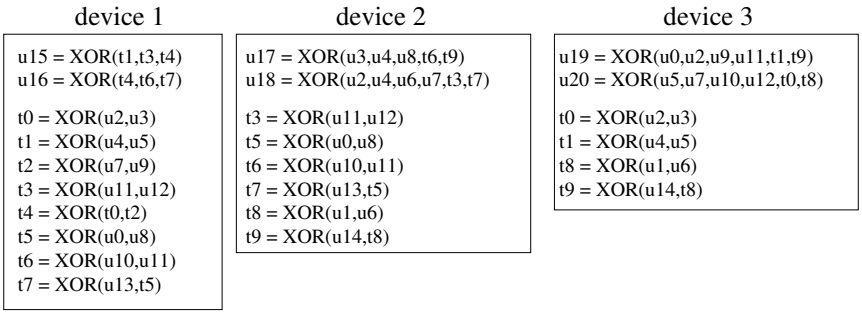

Figure 1: Equation-based load distribution on several OpenCL devices.

5 Performance Evaluation

We measured the coding throughput for several systems (CPU and GPU on two desktop systems and also on a mobile computer for reference purposes, see Table 1) with varying parameters. Note though that the GPU of the mobile system is not capable of running OpenCL. This is the reason why for this system only the CPU is used and only the single device measurements were taken. Similarly, although system 2 has a GPU installed that is compatible to OpenCL it could not be used due to driver issues. As CPU driver the ATI Stream SDK 2.2 (OpenCL 1.0) was applied. All three systems together were only used to show differences between different implementations or when using multiple devices together. In all other tests system 1 was used to study differences between coding styles or parameter sets. For the measurements a small test framework was built around the OpenCL

ID	CPU	RAM	GPU
1	Intel Core2 Quad Q6600@3.0GHz	4GB@400MHz	NVIDIA GeForce 8800GT
2	Intel Core2 Quad Q6600@3.2GHz	4GB@533MHz	not OpenCL capable
3	Intel Core Duo T2300@1.66GHz	1GB@266MHz	not OpenCL capable

Table 1: Test systems

kernels and their invocation code to measure the performance. Data for the coder is generated randomly on the fly. Each test was run 10 times.

CPU performance: The results of comparing the performance with a sequential CPU-based parity encoder (written in C language) and the OpenCL encoder using only the CPU are shown in Figure 2. For comparison, the GPU performance on

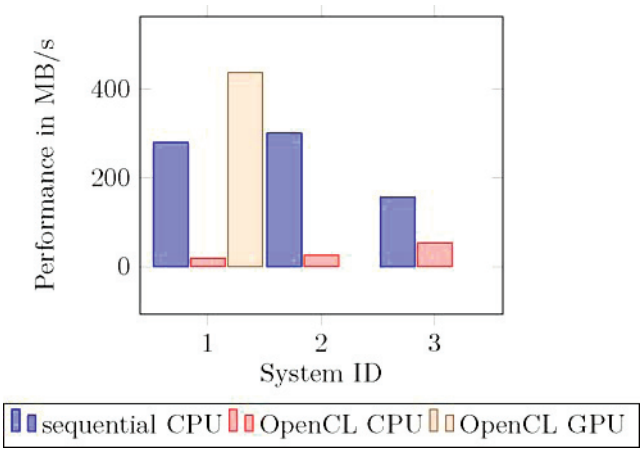

Figure 2: Performance of different CPU implementations. A direct 5+1 coding scheme was used with a stripsize of 1048416 Bytes and a data size of 10^9 Bytes.

system 1 is included. It can be seen that the heavily multithreaded execution by the OpenCL program is not suitable at all for computing such computationally lightweight codes as a simple parity on the CPU. More complex codes like $5 + 2$ which involve $\omega \geq 3$ sub-units perform substantially better on the CPU (see Figure 3) which suggests that the threading overhead by far makes up for the most CPU usage. Figure 2 shows another unexpected result: For less computationally complex codes like $5 + 1$ when only run on the CPU the OpenCL coder on system 3 outperforms both system 2 and 1, in spite of their superior hardware parameters.

Single device versus multi device: Our OpenCL coder performs significantly worse than expected when run on the GPU. As such, running tasks on the GPU does seem to hinder the performance obtained through computations on the CPU. Figure 3 shows that using both CPU and GPU together on system 1 performs worse than running computations using only either one. Under certain circumstances using both CPU and GPU does perform better but these were niche settings that do not reflect the average behavior. Surprisingly, the CPU variant on system 1 outperforms system 2 this time, but although being reproducible, this situation did not occur for other tests where system 2 performed slightly better due to the higher CPU base clock frequency.

Influence of stripsize: Every transfer is associated with overhead. Thus, batching many small transfers into one larger transfer is advised [NVI09]. Correspondingly, a larger stripsize should improve the performance. It was measured that waiting for data transfers to finish takes upon approximately 12% of the time needed to compute a stripe for a 5+2 encoding scheme on the GPU. However, a variation of the strip size did not significantly influence the performance. This might be due to limitations of the GPU driver.

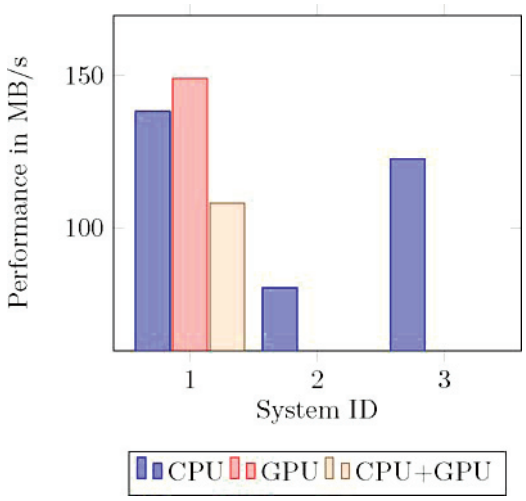

Figure 3: Performance obtained through using different devices. An iterative 5+2 encoding scheme was used with a stripsize of 1048416 Bytes and a data size of 10^9 Bytes.

coding style	code	average [MB/s]	std. deviation [MB/s]
iterative	$20+10$	36.267	0.003361
direct	$20+10$	9.049	9×10^{-6}
iterative	$5+2$	55.35	1.56767
direct	$5+2$	52.96875	4.57461

Table 2: Results for a $20+10$ and a $5+2$ encoding scheme using both the GPU and CPU of system 1, a stripsize of 24kB and a filesize of 10^9.

Iterative versus direct encoding: In order to study the difference between direct and iterative encoding, we calculated a 20+10 coding scheme (see Table 2). Although temporary units are computed by both the CPU and the GPU, the iterative coding scheme still performs better, especially on the more compute intense code (20+10).

Observations: The decision on how to process data in a kernel was heavily influenced by the lightweight threading model present in modern GPU architecture. Judging from the results it might not be the best one for execution on the CPU, though. The OpenCL driver for x86 CPUs spawns one thread per each available core. During the tests these threads together never reached significantly more CPU usage than the equivalent of two processor cores.

For a future work one should generate different kernels for GPUs, CPUs and other accelerator devices - OpenCL allows to differentiate between these three categories and names the IBM Cell Blades as an example for the latter category [ope09].

The NVIDIA display driver for system 1 (see Table 1) spawned one additional

thread on the CPU that had a CPU usage of around 100 percent (equivalent of one busy core). The performance of computations done on the GPU seemed to be dependent on the CPU usage of that thread. Taking CPU time away from that particular thread (the other thread containing the rest of the program only consumed a fraction) decreased the reported performance proportionally to amount of CPU usage taken away by other programs (for example a loop only containing sleep commands of parameterized length). It remains for a future work to determine whether this is a limitation of the graphics driver or of the coder.

6 Summary

OpenCL is an appropriate platform to implement erasure-tolerant coding for storage and memory systems. An equation-based specification of the Cauchy Reed/-Solomon coding procedures can be easily translated to OpenCL code. OpenCL provides data parallelism for processing a high number of data elements in parallel using the same computation algorithm. We could show that coding on a GPU outperforms slightly the CPU. By managing several OpenCL devices (if present), computations can be distributed on them. At present, multiple device computations did not reach the expected higher performance due to device driver limitations.

References

[BBBM95] M. Blaum, J. Brady, J. Bruck, and J. Menon. EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures . *IEEE Transactions on Computers*, 44(2), February 1995.

[BE09] A. Brinkmann and D. Eschweiler. A Microdriver Architecture for Error Correcting Codes inside the Linux Kernel. In *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2009*. ACM, 2009.

[BKK+95] J. Blomer, M. Kalfane, M. Karpinski, R. Karp, M. Luby, and D. Zuckerman. An XOR-based Erasure–resilient Coding Scheme. Technical Report TR-95-048, International Computer Science Institute, August 1995.

[CSWB08] M. L. Curry, A. Skejellum, H. L. Ward, and R. Brightwell. Acellerating Reed-Solomon Coding in RAID Systems with GPUs. In *Proceedings of the 22nd IEEE Int. Parallel and Distributed Processing Symposium*. IEEE Computer Society, 2008.

[HKS+02] A. Haase, C. Kretzschmar, R. Siegmund, D. Müller, J. Schneider, M. Boden, and M. Langer. Design of a Reed Solomon Decoder Using Partial Dynamic Reconfiguration of Xilinx Virtex FPGSs - A Case Study. 2002.

[HSM08] V. Hampel, P. Sobe, and E. Maehle. Experiences with a FPGA-based Reed/Solomon-encoding coprocessor. *Microprocessors and Microsystems*, 32(5-6):313–320, August 2008.

[IR60] G. Solomon I. Reed. Polynomial Codes over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics [SIAM J.]*, 8:300–304, 1960.

[KGP89] R. Katz, G. Gibson, and D. Patterson. Disk System Architectures for High Performance Computing. In *Proceedings of the IEEE*, pages 1842–1858. IEEE Computer Society, December 1989.

[Khr09] Khronos. IBM Releases OpenCL Drivers for POWER6 and Cell/B.E. http://www.khronos.org/news/permalink/ibm-releases-opencl-drivers-for-power6-and-cell-b.e/, October 2009.

[NVI09] NVIDIA. OpenCL Best Practices Guide (version 1.0). page 9. July 2009.

[ope09] The OpenCL Specification (version 1.0, document revision 48), October 2009.

[Pla07] J. S. Plank. Jerasure: A Library in C/C++ Fasciliating Erasure Coding to Storage Applications. Technical Report CS-07-603, University of Tennessee, September 2007.

[PLS+09] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Zooko Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries for Storage. In *FAST '09: Proccedings of the 7th conference on File and storage technologies*, pages 253–265. USENIX Association, 2009.

[SH07] P. Sobe and V. Hampel. FPGA-Accelerated Deletion-tolerant Coding for Reliable Distributed Storage. In *ARCS 2007 Proceedings*, pages 14–27. LNCS, Spinger Berlin Heidelberg, 2007.

[Sob03] P. Sobe. Data Consistent Up- and Downstreaming in a Distributed Storage System. In *Proc. of Int. Workshop on Storage Network Architecture and Parallel I/Os*, pages 19–26. IEEE Computer Society, 2003.

[SP06] P. Sobe and K. Peter. Comparison of Redundancy Schemes for Distributed Storage Systems. In *5th IEEE International Symposium on Network Computing and Applications*, pages 196–203. IEEE Computer Society, 2006.

[SP08] P. Sobe and K. Peter. Flexible Parameterization of XOR based Codes for Distributed Storage. In *7th IEEE International Symposium on Network Computing and Applications*. IEEE Computer Society, 2008.

[SPB10] T. Steinke, K. Peter, and S. Borchert. Efficiency Considerations of Cauchy Reed-Solomon Implementations on Accelerator and Multi-Core Platforms. In *Symposium on Application Accelerators in High Performance Computing 2010*, Knoxville, TN, July 2010.