Efficient AMG on Heterogeneous Systems

Jiri Kraus and Malte Förster

Fraunhofer Institute for Algorithms and Scientific Computing SCAI
Schloss Birlinghoven, 53754 Sankt Augustin, Germany

Abstract. In many numerical simulation codes the backbone of the application covers the solution of linear systems of equations. Often, being created via a discretization of differential equations, the corresponding matrices are very sparse. One popular way to solve these sparse linear systems are multigrid methods - in particular AMG - because of their numerical scalability. As the memory bandwidth is usually the bottleneck of linear solvers for sparse systems they especially benefit from high throughput architectures like GPUs. We will show that this is true even for a rather complex hierarchical method like AMG. The presented benchmarks are all based on the new open source library LAMA and compare the run times on different GPUs to those of an efficient OpenMP parallel CPU implementation. As the memory access pattern is especially crucial for GPUs we have a focus on the performance of different sparse matrix formats.

Keywords: LAMA, AMG, GPU, CUDA, OpenCL

1 Introduction

In this paper we show that it is possible to gain a significant performance increase when GPUs are used for the solution phase of AMG. To achieve this it is necessary to execute the full AMG cycle on a GPU with massively parallel components. By using Jacobi smoothing the full AMG cycle essentially cuts down to a series of sparse matrix vector multiplications (SpMV). So it is possible to achieve good AMG performance for the solver phase if we have good SpMV performance. We show that the popular CSR format does not lead to acceptable performance on GPUs. Instead, more GPU-suitable formats like ELLPACK or JDS are needed. ELLPACK has been successfully used for a GPU AMG implementation by Feng and Zeng[8]. Also Haase, et. al. have been successfully implemented a AMG for GPUs using the interleaved compressed row storage format[10] which is quite similar to our JDS implementation. In contrast to their publications we focus on well known model problems to report comprehensible results. This has been already the approach for our last publication "Scalable parallel AMG on ccNUMA machines with OpenMP"[9], where we have compared our AMG implementation to the open source solver packages PETSc and hypre[2, 1]. This CPU implementation also is the baseline for our GPU benchmarks. To make it easier to classify our results this paper follows the structure of our aforementioned paper and uses the same hardware and model problems.

We start with a short introduction of LAMA the library we used for our AMG implementation. In the following section 3 we describe our hard- and software setup and the execution environment we used. After describing the model problems, the used AMG Setup and the used sparse matrix formats we present the obtained benchmark results. The benchmarks evaluate how the performance is influenced by the matrix storage format, the precision of the calculations, ECC memory protection, the usage of the texture cache on the GPU and the performance impact of OpenCL.

2 LAMA

The Library for accelerated math applications, LAMA, is a new open source project which is available at `http://www.libama.org`. The first of two main design aims of LAMA is to allow easy integration of accelerators like GPGPUs. As a consequence to this the second main design aim is to be extensible with new matrix storage schemes while supporting a natural mathematical syntax without sacrificing performance, like it is also achieved by the C++ Library Blitz++[3]. To achieve both goals LAMA is separated into two parts. A C library which provides BLAS functionality for dense and sparse types and which is used to utilize all types of accelerators and a C++ part which supports the extensibility and provides the natural mathematical syntax. The C library makes our core algorithms of our library usable by a wide range of applications and allows the integration of existing BLAS Libraries. The C++ part uses simplified expression templates [15] to achieve the second design aim. Utilizing this and by formulating solvers only in terms of simple BLAS operations, like they are printed in text books[6], we achieve very comprehensible solver implementations and it is easy to experiment with new accelerators or data structures, e.g. different sparse matrix formats. The results obtained in this paper have been produced with a version of LAMA that is mainly using "compile time polymorphism" through templates. This enables aggressive compiler optimizations while sacrificing some run time flexibility.

Using LAMA we had only a minimal implementation effort to make the AMG implementation, that we also used in our previous publication, run with CUDA and OpenCL. It was only necessary to implement SpMV for the tested sparse matrix storage formats within the back ends for CUDA and OpenCL. In addition we also implemented specializations of the Jacobi smoother for the tested sparse matrix storage formats for both back ends. Although this would not have been necessary from a functional point of view, since we also implemented a universal Jacobi based on just SpMV, the specialized version is slightly more efficient. Because we have also used a specialized version of the Jacobi smoother in the OpenMP back end this was necessary to have a fair comparison.

3 Hardware and Software Setup

The CPU results presented in this paper have been computed on the hardware described in table 1. This is one of the systems we have used in our previous publication [9], where it was named BULL. All binaries have been build with gcc version 4.4.3 and the optimization options `-O3` and `-ffast-math`.

3.1 GPUs

The GPU benchmarks have been done with the GPUs that are listed in table 2. We have used CUDA and the OpenCL implementation from NVidia in version 3.2 for the GPU benchmarks. To compile the GPU kernels we have used the options `-arch=sm_13` and `-use_fast-math` in all cases. If not otherwise mentioned all measurements have been done in double precision with disabled ECC and enabled Texture cache for the access to the input vector in SpMV operations and Jacobi iterations.

Table 1. CPU Hardware

name	CPU
cpu	Xeon X5650
core freq.	2.67 GHz
L3-cache	12 MB
cores/cpu	6
HT	off
sockets	2
cores	12
memory	12 GB
channels	3
mem. type	DDR3
mem. freq.	1333 MHz

Table 2. GPUs used

name	G46	G48	T10	T20
device	GeForce GTX 460	GeForce GTX 480	Tesla C1060	Tesla C2050
compute cap.	2.1	2.0	1.3	2.0
multiprocessors	7	15	30	14
cores	336	480	240	448
core freq	1.43 GHz	1.40 GHz	1.30 GHz	1.2 GHz
memory	1 GB	1.5 GB	4 GB	3 GB
width	256-bit	384-bit	512-bit	384-bit
freq	1.8 GHz	1.8 GHz	0.8 GHz	1.8 GHz
HW Cache	yes	yes	no	yes
ECC	no	no	no	yes

An introduction to CUDA or GPU programming can be found in [11]. We just want to highlight that in contrast to a CPU a GPU is designed as a high throughput architecture. This has certain implications for the performance characteristics of these devices. Because the algorithm under examination is memory bound the most crucial point for us is the high memory bandwidth of GPUs. This high memory bandwidth comes at the cost of a high access latency and strict coalescing requirements to achieve the full memory bandwidth[7]. To overcome the high latency a GPU can manage a lot more threads concurrently than there are compute cores. If enough threads are available it is possible to hide the high access latency to the GPU memory, by switching between threads that are waiting and threads that are ready to run.

To achieve good performance on a GPU this means that it is necessary to have a high degree of parallelism and regular memory accesses. For the chosen solver the high degree of parallelism is given. The regular memory access however is dependent on the chosen storage format for sparse matrices. The influence of the sparse matrix format on the coalescing is described in section 5.1.

4 Execution

All benchmarks have been computed via the benchmark framework integrated in the LAMA package. This framework running in Python ensures reproducible run times by creating new processes for every test run, eliminating the possibility of a benchmark influencing its followers with respect to memory usage. Within every benchmark process, the reported run time is the minimum of 5 executions. Additionally, every process is started at least 3 times. In case aberrations have to be eliminated here this number will automatically increase. Since GPUs do not support preemption and therefore deliver reproducible results this feature is in general only triggered within CPU benchmarks.

4.1 Model Problems

Our set of test matrices is shown in table 3. It consists of different discretizations of the Laplacian operator on structured grids in up to three dimensions. All matrices have a total of 1 million rows but increase in the number of nonzero entries. Each row corresponds to exactly one grid point and its nonzero values refer to the entries of the differential stencil applied. We have chosen these model problems because they are well known, which makes it more easy to compare our results[7]. Additionally, they are a good measure for real world 1D, 2D and 3D applications because of the basic local access patterns common for matrices based on a wide range of PDE applications.

Table 3. Laplacian discretizations used for solver benchmarks

name	dimensions	diags	entries	CSR mem
1D3P	1,000,000	3	3 Mio.	38 MB
2D5P	1,000x1,000	5	5 Mio.	61 MB
3D7P	100x100x100	7	7 Mio.	83 MB
2D9P	1,000x1,000	9	9 Mio.	107 MB
3D27P	100x100x100	27	27 Mio.	307 MB

To exploit the sparsity all matrices are stored in either Compressed Sparse Row (CSR), Jagged Diagonal Storage (JDS) or ELLPACK format using both, single and double precision. More details about the Matrix formats will be given in Section 5.

4.2 AMG Setup Phase

Due to its complexity and partially sequential nature, the setup phase of AMG in LAMA is computed on the CPU. This includes coarse grid definitions, interpolation and restriction constructions as well as the multiplication of the galerkin operators. As a coarsening strategy we use the classical Ruge-Stüben algorithm[14] (1stage) in combination with standard interpolation.

Table 4 shows the galerkin operator stats of the resulting hierarchies for the corresponding 2D and 3D stencils.

Table 4. Galerkin Operator stats for 2D and 3D stencils

	2D5P		2D9P		3D7P		3D27P	
Lvl	Rows	Entries	Rows	Entries	Rows	Entries	Rows	Entries
0	1000000	4996000	1000000	8988004	1000000	6940000	1000000	26463592
1	500000	4492002	250000	6220036	500000	9320600	125000	13642048
2	125000	3105014	62500	2776594	83331	6321285	14456	2762872
3	31250	1380362	15625	684745	10458	1611064	1317	318939
4	7813	338689	3126	120954	966	171576	181	25235
5	1563	59057	601	21161	133	13507	-	-
6	312	10388	121	3405	-	-	-	-
7	60	1452	-	-	-	-	-	-

The coarsening rates, and therefore also the number of levels constructed, are strongly related to the stencil size as well as the problem dimension. These levels in return will define the amount and shape of the SpMV operations used within each AMG V-cycle in the solution phase later on.

Besides the classical meaning of an AMG setup phase our solver initialization also includes the setup of the coarsest grid inverse as well as the needed data conversions and transfers for the GPU devices. Since this whole process is currently only partially parallelized and optimized, we will not consider it for the benchmarks but focus on the solution phase.

4.3 AMG Solution Phase

Here we describe the implementation of the AMG solution phase which is basically a series of SpMV operations. For the benchmarks we measure 10 iterations of a CG solver preconditioned with AMG. In the solution phase AMG is running a V-cycle performing two pre- and post-smoothing steps with a weighted Jacobi.

Although it has no effect on the performance analysis later on, table 5 exemplary shows the convergence history of the resulting AMG approach applied to the 2D9P stencil.

Please keep in mind that we only measure the run times of the solution phase. The transfer of the matrices to GPU memory is considered to be a part

Table 5. L2-residual reduction for 2D9P

Iter	0	1	2	3	...	8	9	10
LAMA	$1.9E+2$	$2.6E+1$	$2.4E+0$	$1.8E-1$	...	$3.2E-7$	$3.5E-8$	$1.7E-9$

of the setup. Theoretically, these transfer costs could also be hidden behind the computation of subsequent level operators. Besides that, also the transfer of the rhs and the solution is not considered in the given run times. This is because they remain constant for a given problem size, independently of the number and complexity of the AMG cycles performed. For our benchmarks the transfer costs for the needed uploads of right hand side and first guess as well as the download of the solution are given in Table 6. The transfer times have been measured with paged locked host memory that enables dma transfers and is necessary to allow asynchronous transfers.

Table 6. Transfer cost of rhs, 1st guess and solution

	float	double
Transfer	$2,4ms$	$4,8ms$
Bandwidth	$4.65GB/s$	$4.65GB/s$

This shows that even with comparably small amounts of data transferred one can utilize a quite satisfying percentage of the maximal available PCI-Express bandwidth of $6GB/s$.

5 Matrix Formats

There are many different storage formats available for sparse matrices. In this paper we will focus on three of them which are quite diverse in their advantages and disadvantages. They will be introduced briefly by showing the main storage vectors for the test matrix in figure 1.

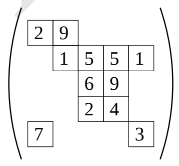


Fig. 1. Example 5x5 matrix

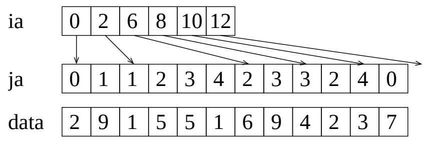


Fig. 2. The CSR storage format

5.1 The CSR Format

The first and probably most commonly used format (at least on CPUs) is the compressed sparse row format (CSR). It keeps all matrix data in two index arrays *ia* and *ja*, as well as the actual matrix values in *data*. The array *ia* keeps track of the start and end of each row in the other two arrays as shown in figure 2, while *ja* and *data* give the column index and value of each nonzero element. To ensure fast access to the diagonal elements of the matrix they are always stored first in each row.

The storage amount is fixed by the number of rows and non zeros and there are no additional requirements for the matrix pattern. Therefore the CSR format is one of the most universal sparse formats available.

To give a baseline of our GPU AMG implementation we compare the run times in single and double precision to the results obtained on the CPU for the popular CSR format. The presented double precision run times on the CPU have been validated against PETSc and boomer AMG from the hypre package[2, 1]. This has been done in our aforementioned publication[9]. Table 7 list the run times for all model problems and for all the hardware mentioned in the tables 1 and 2 in single and double precision. The first column shows the identifier of the tested hardware in this column S means a serial run, 1 means a run that uses one socket (6 Cores) of the CPU system and 2 means a run that uses two sockets (12 Cores) of this system. The other identifiers are the GPUs from table 2.

Table 7. Execution Times for the CSR format in seconds

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
S	1.15	1.17	1.78	1.87	2.01	2.13	2.60	2.86	4.09	4.70
1	0.34	0.58	0.53	0.86	0.62	0.98	0.78	1.25	1.30	1.99
2	0.18	0.32	0.27	0.51	0.32	0.53	0.41	0.67	0.67	1.05
G46	0.11	0.17	0.77	0.96	1.40	1.75	1.98	2.39	6.02	6.45
G48	0.08	0.12	0.51	0.62	0.94	1.10	1.28	1.53	3.81	4.06
T10	0.20	0.30	0.74	0.90	1.14	1.29	1.52	1.63	2.93	3.00
T20	0.09	0.13	0.60	0.74	1.08	1.33	1.54	1.84	4.55	4.87

As one can see from table 7 the execution times for the GPUs are dramatically increasing with the complexity of the model problems. To understand that remember the implementation of the AMG solver phase, which is described in section 4.3. As described there it mainly consist of SpMV operations and is therefore memory bound. Given that the dramatic performance drop for more complex problems can be easily explained by the fact that the memory system of the tested GPUs only runs at full speed if memory accesses can be coalesced[7].

This is the case if 16 neighboring threads are accessing 16 consecutive 4 byte memory locations. In this situation all 16 memory accesses can be carried out in one transaction instead of 16.

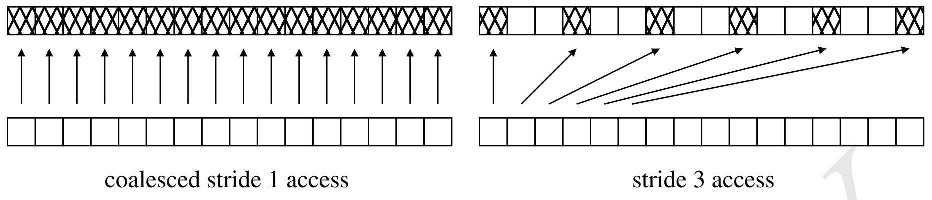


Fig. 3. Memory coalescing

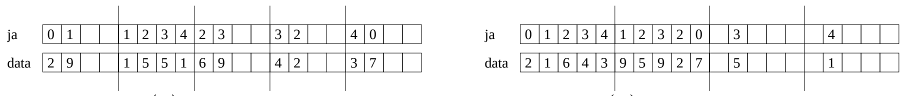
To explain the impact on a CSR SpMV operation let k be the number of none zeros per matrix row. To carry out the SpMV on a GPU one thread for each matrix row is started. Because the matrix is stored in row major order neighboring threads only access consecutive values of the matrix if $k = 1$. The GPUs are able to coalesc memory accesses also for larger strides but then only $\frac{1}{k}$ of the available memory bandwidth is utilized[4]. This is shown in figure 3 for stride $k = 1$ and stride $k = 3$ which corresponds to the model problems 1D3P. So the usable memory bandwidth decreases with the complexity of the problem.

An interesting observation is that the GeForce GTX 480 is slower than the Tesla C1060 for the 3D27P model problem in all cases with the CSR format. Because the GTX 480 has a hardware cache, a much better double precision performance and a higher memory bandwidth this is not as expected. We can only speculate about the reasons, but it is quite reasonable that the hardware cache of the GTX 480 simply is to small. Because of this it is never a benefit because all cache lines are invalidated before they are accessed a second time. So the cache is just overhead in the access to the memory system and gives the Tesla C1060 a small advantage in this case.

5.2 The ELLPACK Format

Looking towards GPUs or vector-processors in general one needs to ensure more regularity in memory access in order to achieve good performance. The sparse format provided by the ELLPACK package[5] ensures easier storage under the additional requirement of an equally number of non zeroes per matrix row. Because of this assumption it does not need the array ia but might artificially increase the number of non zeroes as shown in figure 4(a).

To allow coalesced memory access of in terms of multiple threads reading a sequence of matrix rows at once it is also beneficial to store the nonzero Elements of the matrix column-wise as shown in figure 4(b), which is how ELLPACK is stored on GPU devices for the benchmarks in this paper.



(a) row-wise (b) column-wise

Fig. 4. The ELLPACK storage format

While the number of additional artificial elements needed to meet the storage requirements for ELLPACK might not be high for the actual system matrix created from a differential stencil of some specific pattern, this does not need to hold when looking at the whole AMG hierarchy of matrices. Especially Interpolation operators are usually very unbalanced in terms of nonzero entries. Table 8 shows the overall overhead of artificial non zeros throughout the AMG hierarchy of matrices.

Table 8. Storage overhead of ELLPACK versus CSR for the AMG hierarchy

Stencil	1D3P	2D5P	2D9P	3D7P	3D27P
overhead	14%	14%	11%	17%	13%

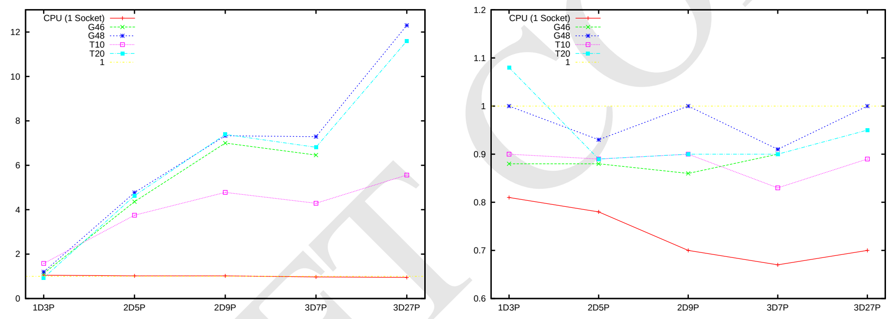
The run times of the benchmarks with the ELLPACK format are given in table 9. The table is formatted like table 7 in section 5.1. The run time for the model problems 3D27P in double precision on the GeForce GTX 460 is missing because the available global memory on this device is not large enough store the whole solver setup in addition to the texture memory reserved by the desktop environment.

Table 9. Execution Times for the ELL format in seconds

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
S	0.97	1.12	1.62	1.87	1.90	2.17	2.51	2.95	4.10	4.74
1	0.31	0.55	0.52	0.84	0.61	0.96	0.81	1.29	1.36	2.10
2	0.15	0.27	0.27	0.44	0.31	0.53	0.42	0.66	0.70	1.07
G46	0.10	0.15	0.16	0.22	0.19	0.25	0.26	0.37	0.43	N/A
G48	0.07	0.10	0.10	0.13	0.11	0.15	0.16	0.21	0.25	0.33
T10	0.11	0.19	0.15	0.24	0.17	0.27	0.25	0.38	0.38	0.54
T20	0.08	0.14	0.12	0.16	0.13	0.18	0.19	0.27	0.30	0.42

As one can see from figure 5(a) all tested GPUs have a huge benefit from the ELLPACK format. This can be explained with the same arguments like the bad performance of the CSR format on GPUs. Because we use a column major order layout of the ELLPACK format on the GPU all accesses to the input matrix are perfectly coalesced and therefore no memory bandwidth is wasted[7]. For the CPU version of ELLPACK we are using row major order storage to have a good cache utilization while accessing the input matrix. Although the performance of ELLPACK remains constant its speedup increases because the CSR performance is decreasing for the more complex system like it has been described in section 5.1.

The CPU run times are nearly not affected by the storage format because the additionally stored matrix elements of the ELLPACK format are compensated by fewer indirect memory accesses and the fact that we can save the storage of one integer array. It would be possible to further optimize the ELLPACK format on the CPU, but this has not been done because it was not in the focus of this work.

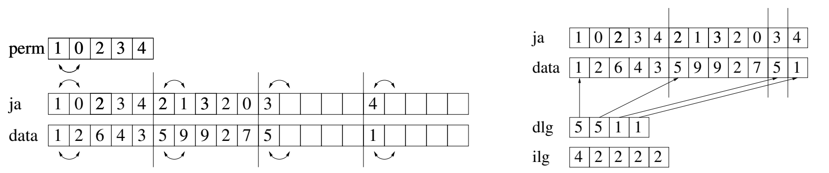


(a) ELLPACK vs. CSR (b) JDS vs. ELLPACK

Fig. 5. Speedup between Storage formats in double precision (colors on line)

5.3 The JDS Format

For many applications the additional storage of artificial zeroes are a knock-out criterion for the ELLPACK format. For matrices with few long rows this overhead will turn out to be much higher than the percentage given in table 8. To have a more general matrix storage structure we will also look at the Jagged Diagonal Sparse (JDS) format, which has been known to perform well on graphics cards as shown in [12]. The format is a compromise between highly efficient coalesced memory accesses on GPUS from ELLPACK and the flexibility of CSR.



(a) permutation of ELLPACK (b) final arrays

Fig. 6. The JDS storage format

The arrays for JDS are very similar to the column-wise ELLPACK storage in figure 4(b). To remove the artificial elements we have to reorder the rows by size and store the permutation in the array *perm*. Now this permutation is also applied to every column in the arrays *ja* and *data* as shown in figure 6(a), sorting all matrix entries to the front of every column.

Given an extra array for each column size *dlg*, it is now safe to remove all artificial values. Additionally we store an array *ilg* for the number of elements in each row which will give us easier access in certain loops. Note that in general only one of the arrays *dlg* and *ilg* is needed, the second one is purely optional. Figure 6(b) shows the modified arrays for JDS.

Looking at the run times of JDS in table 10 the first thing to notice are the comparably bad timings on the cpu. While there was a row-wise implementation for ELLPACK on the CPU we only support column-wise ordering for JDS. Of course this results in rather bad memory access patterns on a non-vector architecture. In comparison to ELLPACK there are less matrix elements to load within the JDS format, but this comes at the cost of two additional vectors *perm* and *dlg*. This has negative influence on the performance especially on the Tesla C1060, since the card has no fast cache for those arrays. All graphics cards based on the Fermi architecture can compensate for these with the reduced load in *ja* and *data*. Note that the reduced overall memory makes it even possible to run the larger $3D27P$ example in double precision on the GeForce 460 which was not possible with ELLPACK.

Comparing the run times of the JDS directly to ELLPACK we see only slight differences as shown in figure 5(b). As mentioned before only the 'cacheless' C1060 suffers from the more general storage format. For all other cards the differences are tolerable. Nevertheless JDS is much more flexible than ELLPACK looking at general matrix patterns with diverse row lengths that would lead to much higher padding overhead.

6 Single precision vs double precision performance

The run times with single precision arithmetic are given in tables 7, 9 and 10. A rationale for these numbers can again be derived from the characteristics of a SpMV. If we state that

Table 10. Execution Times for the JDS format in seconds

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
S	1.27	1.41	3.45	3.91	4.40	5.99	7.02	8.06	10.04	12.14
1	0.37	0.62	0.77	1.08	0.98	1.37	1.5	1.92	2.27	3.01
2	0.21	0.33	0.42	0.59	0.55	0.81	0.85	1.07	1.23	1.56
G46	0.10	0.17	0.15	0.25	0.17	0.29	0.26	0.41	0.40	0.62
G48	0.08	0.10	0.10	0.14	0.11	0.15	0.17	0.23	0.24	0.33
T10	0.13	0.21	0.18	0.27	0.21	0.30	0.32	0.46	0.45	0.61
T20	0.09	0.13	0.11	0.18	0.13	0.20	0.20	0.30	0.29	0.44

– the execution time for SpMV is limited by the memory bandwidth
– we ignore the existence of caches.

we can calculate the possible speedup of single precision calculation over a double precision calculation for a SpMV theoretically.

To do that, let A be our input matrix with n rows, n columns and for simplicity k none zero elements per row. To do a SpMV depending on the storage format the following values need to be accessed:

CSR	ELL	JDS	
$n \cdot k$	$n \cdot k$	$n \cdot k$	acc. to the input vector
n	n	n	acc. to the output vector
$n \cdot k$	$n \cdot k$	$n \cdot k$	acc. to the none zero elements of A
$n \cdot k$	$n \cdot k$	$n \cdot k$	acc. to the column index array of A
n	0	0	acc. to the row index array of A
0	0	n	acc. to the permutation array of A
0	0	$n \cdot k$	acc. to the *dlg* index array of A
$n \cdot (2 \cdot k + 1)$	$n \cdot (2 \cdot k + 1)$	$n \cdot (2 \cdot k + 1)$	acc. to floating point values
$n \cdot (k + 1)$	$n \cdot k$	$n \cdot (2 \cdot k + 1)$	acc. to integer values

With the size of a single precision floating point value being $4b$, a double precision value $8b$ and a integer value $4b$ this leads to

CSR	ELL	JDS	
$n \cdot (5 \cdot k + 3) \cdot 4$	$n \cdot (5 \cdot k + 2) \cdot 4$	$n \cdot (6 \cdot k + 3) \cdot 4$	bytes in double precision
$n \cdot (3 \cdot k + 2) \cdot 4$	$n \cdot (3 \cdot k + 1) \cdot 4$	$n \cdot (4 \cdot k + 2) \cdot 4$	bytes in single precision

Because the accesses to the row index array do not grow with the number of none zeros the ratio of double precision to single precision gets bigger with increasing values of k for CSR. For ELL the ratio is getting smaller with increasing values of k and for JDS they remain constant. This leads to the following upper bounds for the theoretical speedup of single precision over double precision.

CSR	ELL	JDS
$\sup_{k>=1} \frac{(5\cdot k+3)}{(3\cdot k+2)} = \frac{5}{3}$	$\sup_{k>=1} \frac{(5\cdot k+2)}{(3\cdot k+1)} = \frac{7}{4}$	$\sup_{k>=1} \frac{(6\cdot k+3)}{(4\cdot k+2)} = \frac{3}{2}$

Taking into account that the peak single precision performance is 8-times of its double precision performance for a Tesla C1060 and 2-times for the other tested GPUs it is obvious that the double precision compute performance is not the bottle neck for a SpMV operation with double precision. That this is not only a theoretical investigation can be seen in figure 7. The fact that the speedup is larger than the stated maximum for the model problem 1D3P can be explained by cache effects (L1/L2/Texture). Because for smaller stencils we have a better cache utilization which makes the theoretical calculation to pessimistic. The more significant single precision performance drop for the CSR format on the GPUs can be explained by the fact that memory coalescing is a lesser issue for double precision, because the doubled size means that only 8 consecutive elements need to be accessed to exploit the available memory bandwidth.

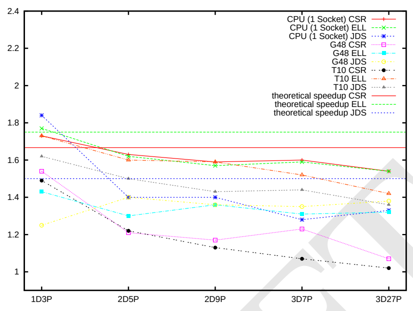


Fig. 7. Speedup of the single precision vs. double precision (colors online)

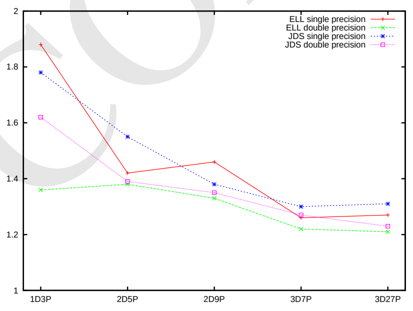


Fig. 8. Overhead of ECC memory protection on Tesla C2050 (colors online)

7 Performance impact of ECC

In table 11 the run times on the GPU Tesla C2050 with and without ECC are given. The rows marked with * are with enabled ECC memory protection. The ECC memory protection can correct single-bit errors and also detect double-bit errors in the DRAM memory, GPU L2 cache, L1 caches and streaming multi-processor registers[13].

As one can see from figure 8 the performance drops by 10 to over 50 percent. The memory bound properties of the AMG solver phase are definitely a reason for this, but because there are no detailed information available how the ECC protection works on Fermi hardware this is speculative.

8 Performance impact of the Texture Cache

Table 12 shows the run times for all model problems on Tesla C1060 and GeForce GTX480 with (+T) and without (-T) the usage of the texture cache to access the input vector in SpMV operations and the Jacobi relaxations. For JDS the index vector *dlg* is also accessed through the texture cache if it does not fit into shared memory.

As one can see from figure 9 the texture cache is slower than the hardware cache of the GTX 480. The seldom cases where the GTX 480 benefits from the the usage of the texture cache can be explained by the fact that more total cache is available. The Tesla C1060 has a huge benefit from the texture cache in case of the ELLPACK and JDS format, because all accesses to the input matrix and the output vectors are perfectly coalesced and the only unstructured access to the input vector therefore benefits from the availability of the cache. One could argue that this effect should be getting smaller for the bigger stencils. This however is not the case because for the smaller stencils parts of the accesses to the input vector can be coalesced so the cache is less important in these cases.

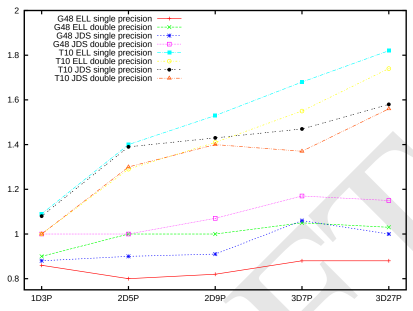


Fig. 9. Speedup using Texture cache (colors online)

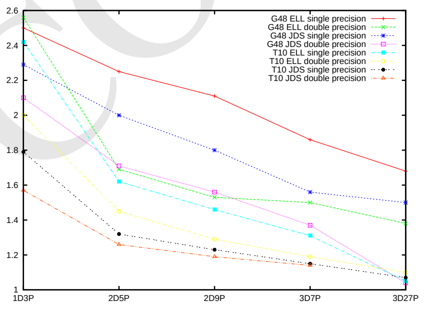


Fig. 10. Speedup of CUDA vs. OpenCL (colors online)

9 CUDA vs. OpenCL Performance

In table 13 the run times for all model problems on the GeForce GTX 480 and the Tesla C1060 are shown for CUDA (C) and OpenCL (O). To give a fair comparison, the texture cache has been disabled in the CUDA version, because there is no efficient way to utilize it within OpenCL.

As one can see in figure 10 the speedup of CUDA is larger for the smaller (lower dimensional) model problems. Looking at a standard AMG cycle it is apparent that 8 kernel launches are required on each level, except the coarsest one, of the AMG hierarchy. These are 4 launches for the 2 pre and post smoothing steps, 1 kernel launch for the interpolation, 1 kernel launch for the

Table 11. Execution Times with and without ECC

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
ELL	0.08	0.14	0.12	0.16	0.13	0.18	0.19	0.27	0.30	0.42
ELL*	0.15	0.19	0.17	0.22	0.19	0.24	0.24	0.33	0.38	0.51
JDS	0.09	0.13	0.11	0.18	0.13	0.20	0.20	0.30	0.29	0.44
JDS*	0.16	0.21	0.17	0.25	0.18	0.27	0.26	0.38	0.38	0.54

Table 12. Execution Times with and without Texture Cache

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
G48 ELL +T	0.07	0.10	0.10	0.13	0.11	0.15	0.16	0.21	0.25	0.33
G48 ELL -T	0.06	0.09	0.08	0.13	0.09	0.15	0.14	0.22	0.22	0.34
T10 ELL +T	0.11	0.19	0.15	0.24	0.17	0.27	0.25	0.38	0.38	0.54
T10 ELL -T	0.12	0.19	0.21	0.31	0.26	0.38	0.42	0.59	0.69	0.94
G48 JDS +T	0.08	0.10	0.10	0.14	0.11	0.15	0.17	0.23	0.24	0.33
G48 JDS -T	0.07	0.10	0.09	0.14	0.10	0.16	0.18	0.27	0.24	0.38
T10 JDS +T	0.13	0.21	0.18	0.27	0.21	0.30	0.32	0.46	0.45	0.61
T10 JDS -T	0.14	0.21	0.25	0.35	0.30	0.42	0.47	0.63	0.71	0.95

Table 13. Execution Times with CUDA and OpenCL

	1D3P		2D5P		2D9P		3D7P		3D27P	
	S	D	S	D	S	D	S	D	S	D
G48 ELL C	0.06	0.09	0.08	0.13	0.09	0.15	0.14	0.22	0.22	0.34
G48 ELL O	0.15	0.23	0.18	0.22	0.19	0.23	0.26	0.33	0.37	*N/A*
T10 ELL C	0.12	0.19	0.21	0.31	0.26	0.38	0.42	0.59	0.69	0.94
T10 ELL O	0.29	0.38	0.34	0.45	0.38	0.49	0.55	0.70	0.77	0.99
G48 JDS C	0.07	0.10	0.09	0.14	0.10	0.16	0.18	0.27	0.24	0.38
G48 JDS O	0.16	0.21	0.18	0.24	0.18	0.25	0.28	0.37	0.36	*N/A*
T10 JDS C	0.14	0.21	0.25	0.35	0.30	0.42	0.47	0.63	0.71	0.95
T10 JDS O	0.25	0.33	0.33	0.44	0.37	0.50	0.54	0.72	0.78	1.02

restriction, 1 kernel launch to set the coarse solution to 0 and 1 kernel launch for the calculation of the residual. On the coarsest level only 1 kernel launch is required. Table 14 shows the total number of necessary kernel launches for each model problem. So it is reasonable to say that the differences between CUDA an OpenCL that are shown in figure 10 can be explained due to a more effective run time of NVidia's CUDA implementation. Because not only the number of kernel launches decreases with the complexity of the model problem, also run time of each kernel increases, this becomes even more evident. The OpenCL run times on the GTX 480 are missing for the model problem 3D27P in double precision, because there have been technical difficulties with our OpenCL version whenever more than half of the available GPU memory was used.

Table 14. Number of kernel launches for one AMG iteration

Probl.	1D3P	2D5P	2D9P	3D7P	3D27P
Num Kernel launches	89	49	41	33	25

10 Conclusion

We have shown that the incorporation of GPUs for AMG can give a performance boost for the solution phase if the right sparse matrix format is chosen. The evaluation of different GPUs shows that this is even true for cheap devices like the GeForce GTX 460. Supplementary we have taken a closer look at the aspect of single precision calculation, L1, L2 and Texture cache on the GPU, the performance impact of ECC and compared the portable approach OpenCL to the proprietary CUDA. In general our results confirm the expectable for memory bound algorithms. Anyhow the huge performance impact of ECC is disturbing. Closing our conclusion we want to accentuate that the theoretical disadvantage for double precision calculations of GPUs is really no issue for memory bound algorithms, like AMG.

11 Future Work

Choosing the right sparse matrix format makes the GPU a really good piece of hardware to compute the solution phase of AMG. But if we take a look at the performance of the whole algorithm the setup phase also has big optimization potential. To address this two things should be done. First the proof that it is possible that the transfer of the AMG hierarchy into GPU memory can be almost completely hidden behind its own computation on the CPU. Given this proof also very sophisticated AMG setups can benefit from GPUs during the solution phase, even if the setup process is to complicated to execute effectively on the

GPU. The second thing to do is to accelerate the setup with the integration of GPUs, by executing parts or even the whole setup on the GPU.

Besides that more techniques to speedup the solution phase should be explored. These include the effect of mixed precision calculation and the option to choose different matrix storage formats for different parts of the algorithm.

Alongside to these AMG related topics we want to support distributed memory machines with LAMA. This will also address multi GPU aspects and the possibility to effectively utilize CPU and GPU resources in parallel.

References

1. hypre homepage. `https://computation.llnl.gov/casc/hypre/software.html`, last viewed Dez 2010, 2010.
2. Petsc homepage. `http://www.mcs.anl.gov/petsc/petsc-as/`, last viewed Dez 2010, 2010.
3. Blitz++ homepage. `http://www.oonumerics.org/blitz/`, last viewed Jan 2011, 2011.
4. Cuda c programming guide. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf`, last viewed April 2011, 2011.
5. Ellpack homepage. `http://www.cs.purdue.edu/ellpack/`, last viewed Apr 2011, 2011.
6. R. Barrett. *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial Mathematics, 1994.
7. N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. In *Proc. ACM/IEEE Conf. Supercomputing (SC), Portland, OR, USA*, 2009.
8. Z. Feng and Z. Zeng. Parallel multigrid preconditioning on graphics processing units (GPUs) for robust power grid analysis. In *Proceedings of the 47th Design Automation Conference*, pages 661–666. ACM, 2010.
9. M. Förster and J. Kraus. Scalable parallel AMG on ccNUMA machines with OpenMP. In *Accepted for Publication on Proc. International Supercomputing Conf (ISC), Hamburg, Germany*, 2011.
10. G. Haase, M. Liebmann, C. Douglas, and G. Plank. A parallel algebraic multigrid solver on graphics processing units. *High Performance Computing and Applications*, pages 38–47, 2010.
11. D. Kirk and W. Wen-mei. *Programming massively parallel processors: A Hands-on approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2010.
12. H. Klie, H. Sudan, R. Li, and Y. Saad. Exploiting capabilities of many core platforms in reservoir simulation. In *SPE Reservoir Simulation Symposium*, 2011.
13. J. Nickolls and W. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, 2010.
14. J. Ruge and K. Stüben. Algebraic Multigrid (AMG). *Multigrid Methods, S.F.McCormick, ed., vol. 3 of Frontiers in Applied Mathematics, SIAM, Philadelphia*, pages 73–130, 1987.
15. D. Vandevoorde and N. Josuttis. *C++ templates: the Complete Guide*. Addison-Wesley Professional, 2003.