Performance and Numerical Accuracy Evaluation of Heterogeneous Multicore Systems for Krylov Orthogonal Basis Computation*

Jérôme Dubois[1] and Christophe Calvin[1] and Serge Petiton[2]

[1] Commissariat l'Energie Atomique,
CEA-Saclay/DEN/DANS/DM2S/SERMA/LLPR
F-91191 Gif-sur-Yvette Cedex, France
[2] Université de Lille 1, Laboratoire d'Informatique Fondamentale de Lille
F-59650 Villeneuve d'Ascq Cedex, France

Abstract. We study the numerical behavior of heterogeneous systems such as CPU with GPU or IBM Cell processors for some orthogonalization processes. We focus on the influence of the different floating arithmetic handling of these accelerators with Gram-Schmidt orthogonalization using single and double precision. We observe for dense matrices a loss of at worst 1 digit for CUDA-enabled GPUs as well as a speed-up of 20x, and 2 digits for the Cell processor for a 7x speed-up. For sparse matrices, the result between CPU and GPU is very close and the speed-up is 10x. We conclude that the Cell processor is a good accelerator for double precision because of its full IEEE compliance, and not sufficient for single precision applications. The GPU speed-up is better than Cell and the decent IEEE support delivers results close to the CPU ones for both precisions.

1 Introduction

In the scientific computing domain, many problems require the computation of an orthogonal basis. The purpose of this technique is to compute an orthogonal basis spanning some linear subspace. The orthogonality of this basis is critical for problems such as solving systems with the GMRES method[16], or computing eigenvalues with the QR method or the Arnoldi process[4]. Depending on the orthogonalization process, the quality of the basis may be impacted due to numerical rounding error. The behavior of this error is predictable among modern mainstream processors essentially because of the IEEE floating arithmetic norm[13] and it is generally taken for granted that mainstream processors are fully IEEE compliant. Consequently the error for elementary floating calculations should be the same for any fully IEEE compliant processors : 10^{-8} in Single Precision (or SP) and 10^{-16} for Double Precision (or DP).

* Candidate to the Best Student Paper Award

CGS	MGS	CGSr
1. for i = 1 : m 2. $v_{i+1} = Av_i$ 3. $H_{1:i,i} = (v_{i+1}, v_{1:i})$ 4. $v_{i+1} = v_{i+1} - H_{1:i,i}.v_{1:i}$ 5. $H_{i,i+1} = \|v_{i+1}\|_2$ 6. $v_{i+1} = \frac{v_{i+1}}{H_{i,i+1}}$ 7. end	1. for i = 1 : m 2. $v_{i+1} = Av_i$ 3. for j = 1 : i 4. $H_{j,i} = (v_{i+1}, v_j)$ 5. $v_{i+1} = v_{i+1} - H_{j,i}.v_j$ 6. end 7. $H_{i,i+1} = \|v_{i+1}\|_2$ 8. $v_{i+1} = \frac{v_{i+1}}{H_{i,i+1}}$ 9. end	1. for i = 1 : m 2. $v_{i+1} = Av_i$ 3. $H_{1:i,i} = (v_{i+1}, v_{1:i})$ 4. $v_{i+1} = v_{i+1} - H_{1:i,i}.v_{1:i}$ 5. $C_{1:i,i} = (v_{i+1}, v_{1:i})$ 6. $v_{i+1} = v_{i+1} - C_{1:i,i}.v_{1:i}$ 7. $H_{1:i,i}$ += $C_{1:i,i}$ 8. $H_{i,i+1} = \|v_{i+1}\|_2$ 9. $v_{i+1} = \frac{v_{i+1}}{H_{i,i+1}}$ 10. end

Table 1. Classical Gram-Schmidt(CGS), Modified G-S(MGS), CGS with reorthogonalization(CGSr)

Emerging computing architectures do not always fully respect this IEEE norm. It is the case of first and second[3] generation NVidia CUDA-enabled GPUs and the STI Cell processor. The upcoming Fermi/GT4xx architecture from NVidia will fully support the recent IEEE 754-2008, respecting a more recent standard than modern processors. In this aspect, the study described in this paper still applies for this upcoming hardware. The two accelerated architectures offer very high achievable computing power compared to classical multicore CPUs, for less Watts / GFLOPs. The peak computing power of the Cell is 200 GFLOPs in single precision(SP) and 100 GFLOPs in double precision(DP). For the best 2009 scientific CUDA-enabled GPU, it is 933 GFLOPs for SP and almost 100 GFLOPs for DP. Also, the memory bandwidth of GPUs varies from dozens of GBytes/s to almost 200 GBytes/s. It is more than any mainstream CPU-based system can achieve.

With the purpose of orthogonalizing faster using GPUs or Cell processors, we want in this paper to focus on the influence of the non fully IEEE-compliance of these architectures compared to a fully IEEE compliant CPU in SP and DP. We first test GPU and Cell with dense matrices by using some well known generated matrices from MatrixMarket, and apply the orthogonalization process using manufacturer BLAS[4] [1] routines. We then analyze the results in terms of performance and accuracy and test the sparse case for the GPU.

This paper will be organized as follows. In section 1 we will explain the different orthogonalization algorithms that will be used. In section 2 we will describe the orthogonalization process. Section 3 will focus on the NVidia CUDA-enabled

[3] First generation is hardware 1.0 and 1.1 for G80/90 GPUs, and second generation is Hardware 1.2 and 1.3 for GT200 GPUs.

[4] Basic Linear Algebra Subroutines

GPU and the IBM Cell processor to explain the hardware and IEEE differences with CPUs. Section 4 will present the implementation of the different orthogonalization processes. Finally we will discuss in section 5 the results for dense and sparse matrices in terms of quality of the orthogonal basis and performances.

2 Orthogonalization Process

Several variants of the orthogonalization process exist [11] : Householder reflections, Givens rotations and the Gram-Schmidt process are the most common ones. The Gram-Schmidt process is declined in different versions : classical(CGS), modified(MGS) and classical with reorthogonalization(CGSr) for instance. These algorithms are described in table 1. The classical version is the simplest and easily parallelizable as seen in [17] : it is faster, as it can take advantage of BLAS 2 operations instead of BLAS 1 in the modified algorithm. But the main drawback of CGS is the possibly high loss of orthogonality within the computed basis due to round-off error [9]. The MGS algorithm tries to correct this, and in fact it manages to provide a more accurate version of the Gram-Schmidt process. Mathematically both are the same, but the MGS provides less parallelism. The CGSr process provides an even more accurate version, by reorthogonalizing each computed vector. In practice it is not necessary to reorthogonalize at each iteration, and so the cost of this algorithm using selective reorthogonalization is close to CGS while being more accurate.

Accuracy of the orthogonal basis is a crucial matter when we apply the orthogonalization process for the Arnoldi Iteration. The orthogonality has an impact on the computed eigenvalues [4], as the constructed orthogonal basis is involved in the projection of the Ritz vectors to obtain the eigenvectors.

3 Accelerators programming

In this section we will expose the architecture as well as the programming paradigm of the GPUs and the Cell processor.

3.1 Nvidia CUDA-enabled GPUs

For several years, GPU processing power has kept steadily increasing, outperforming the peak computing power of the best multicore processors [15]. Because of this, researchers started to study the use of GPUs for scientific computations, and one of the most successful attempt was Brook [5], a scientific language to exploit GPUs.

Hardware and language. In 2006 Nvidia, a major mainstream graphics card manufacturer, released a first version of CUDA[5], its programming language for Nvidia G80 and above series. CUDA aims at providing a comprehensive paradigm able to exploit the massive parallelism of a GPU. This language can

[5] Compute Unified Device Architecture

not be used with other manufacturer's graphics cards. The memory is faster than classical central memory and the access pattern must be regular to achieve great performance just like for a vector machine.
IEEE compliance. The hardware IEEE floating point norm is handled slightly differently than on fully IEEE architectures. Here are some examples of these differences for SP only, taken from the programming manual [15] :

- Addition and multiplication are often combined into a single multiply-add instruction (FMAD), which truncates the intermediate result of the multiplication;
- Division is implemented via the reciprocal in a non-standard-compliant way;
- Square root is implemented via the reciprocal square root in a non- standard-compliant way;
- For addition and multiplication, only round-to-nearest-even and round-towards-zero are supported via static rounding modes; directed rounding towards +/- infinity is not supported;

There are more details about IEEE support of Nvidia GPUs in the programming manual [15].

3.2 STI Cell processor

The Cell processor is the product of the joint between Sony, Toshiba and IBM. The objective of the Cell is to provide an efficient novel multicore design for multimedia applications. As for video cards, its original field is far from the scientific computing field. Despite this, it has been improved to fully handle IEEE double precision arithmetic. The fastest supercomputer in the top500 from June 2009 [14] is the Roadrunner from IBM, delivering more than one PetaFLOPs.
Hardware and language. The Cell processor embeds 9 cores : one classical PowerPC core, called PPE[6], and 8 computing cores called SPEs[7]. As a result, the basic paradigm to program the Cell is as follows : one initializes the application using the PPE, and then spawns threads on the SPEs to do or help computations.
IEEE compliance. The PPE is fully IEEE 754 compliant as it is a PowerPC processor. In single precision, SPEs only implement round-towards-zero rounding mode as opposed to the standard round-to-even mode. It can impact the calculations as seen in [8] where 2 to 3 bits of accuracy are lost. The data format follows the IEEE standard 754 definition, but single precision results are not fully compliant with this standard (different overflow and underflow behavior, support only for truncation rounding mode, different denormal results) [2]. The programmer should be aware that, in some cases, the computation results will not be identical to IEEE Standard 754 ones. For double precision, the Cell processor is fully IEEE-compliant.

[6] PowerPC Processing Element
[7] Synergistic Processing Element

4 Optimizations

In this section, we will see the implementation and optimizations applied to the orthogonalization processes for each hardware : reference CPU, CUDA-enabled GPUs and the STI Cell processor.

4.1 BLAS operations

We saw in table 1 the different versions of the Gram-Schmidt process. For each of them, the same basic BLAS operations apply.

- Matrix vector multiplication : gemv
- Vector dot product : dot
- Scaled vector addition : axpy
- Scaling of a vector : scal
- *optionally* : vector norm 2 computation, nrm2. It is the square root of the dot product of the vector.

If we take CGS as an example, then we have :

1. for i = 1 : m
2. $v_{i+1} = Av_i \leftarrow$ one gemv operation
3. $H_{1:i,i} = (v_{i+1}, v_{1:i}) \leftarrow$ several dot products
4. $v_{i+1} = v_{i+1} - H_{1:i,i}.v_{1:i} \leftarrow$ several axpy operations
5. $H_{i,i+1} = |v_{i+1}|_2 \leftarrow$ one norm2 computation
6. $v_{i+1} = \frac{v_{i+1}}{H_{i,i+1}} \leftarrow$ one scaling of a vector
7. end.

Same basic kernels are being used for the other versions of the GS orthogonalization. Consequently, we may use optimized BLAS operations to take advantage of the different hardware in an optimized and portable manner.

4.2 CPU

The implementation of the dense orthogonalization process on classical CPU follows the standard algorithm described in table 1 and uses ATLAS[8] subroutines [18]. ATLAS is a superset of BLAS library, that adapts some parameters to the architecture by probing its caches and performances according to the number of cores used, at library compilation time. In our case all the operations are BLAS 1 or 2. ATLAS implementations of the BLAS 1 and 2 operations use only one thread, and so does our application

Concerning the sparse case, we follow the standard CSR matrix vector product, mixed with ATLAS BLAS 1 routines to comply with the described GS algorithms.

[8] Automatically Tuned Linear Algebra Subroutines

4.3 GPU

Implementation on the GPU for the dense case is also close to the original algorithms and uses mainly CUBLAS [6], except that we have to handle the memory operations between host and device memory : allocation and transfers.

For sparse matrices, we use the optimized sparse matrix vector multiply from NVidia, which can handle different matrix format : COO[9], CSR[10], ELL[11], DIA[12], and hybrid (DIA+ELL). See [3] for more details about the implementation and the hybrid format. Provided this optimized Matrix-Vector product, we use CUBLAS routines to exploit the GPU power in a portable way, following the steps described in table 1.

4.4 Cell Broadband Engine

As for the GPU hardware, the global algorithm was respected for dense matrices, but an accelerated implementation on the Cell processor implied more programming effort. After data initialization on the PPE, the computing part is distributed among the SPEs equally. There is a BLAS interface used locally on each SPE. This way we have an optimized BLAS operation using all the SPE power. The matrix-vector product uses a rowwise computing pattern, using the EIB to stream data between the processing elements. The locally computed part of the vector is used for the vector operations, an data exchanges may be necessary between SPEs.

Concerning sparse matrices, the achieved performance in [19] varies from 2 to 5 GFlops on several matrices for a highly tuned matrix vector product. The speed-up compared to the capabilities of our reference processor would have been less interesting than for the GPU, and so we did not experiment the sparse CGS orthogonalization for the Cell processor.

5 Experimentation

This section will present the results in terms of precision and performance, each time for both the dense and sparse matrices. The hardware characteristics are described in table 2.

5.1 Hardware precision

Experimentation methodology The precision of the GS process for one subspace size is calculated using the largest absolute dot product between vectors of the orthogonal basis :

[9] COOrdinate
[10] Compressed Sparse Row
[11] ELLPACK
[12] Diagonal

Table 2. Hardware used for experimentation. Frequency is in GHertz and memory in GBytes. The bandwidth is the peak memory bandwidth and sust. is the sustainable bandwidth. Both are expressed in GBytes/s. The peak power is expressed in GFLOPs.

Hardware	Frequency	cores	memory	bandwidth / sust.	Peak Power (DP/SP)
Xeon e5440	2.83	4	4	5.33 / 5.0	45.6 / 91.2
Tesla c1060	1.296	320	4	100 / 75	75 / 933
Cell QS22	3.2	8	16	25.6 / 25.6	100 / 200

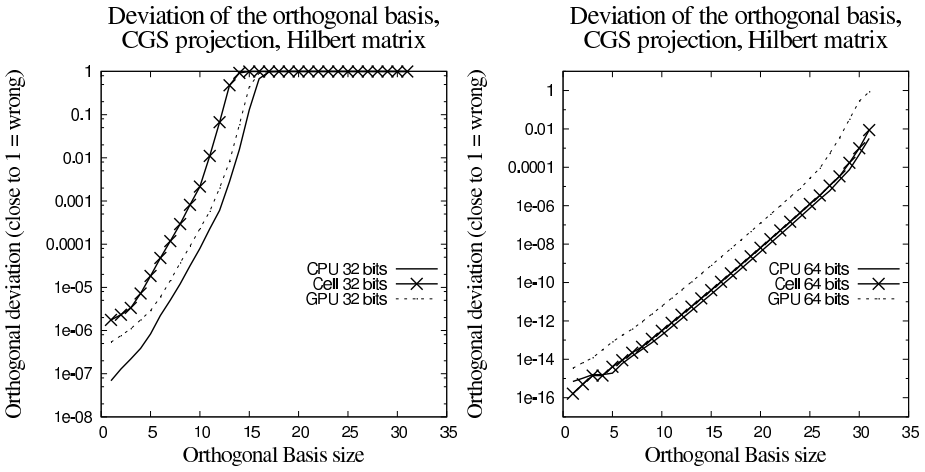

Fig. 1. Precision of the CGS orthogonalization of a 10240 square Hilbert matrix, using different hybrid architectures. The left figure shows the single precision computations, and the right one double precision. The X axis represents the subspace size starting from one, and the Y axis the accuracy of the basis, e.g. the worst orthogonality between two vectors of V.

$$\text{Orthogonal error} : \max |< v_i, v_j >|,\ i \neq j \text{ and } v_i, v_j \in V \text{ basis.}$$

Precisely, this would be the worst orthogonality between two vectors of the basis. Also, the accumulator of the dot products uses the same precision as the rest of the calculations : if the precision is SP, then the accumulator is SP too. Same applies for DP. This way we do not add another parameter to the calculations.

Dense matrices We tested CGS and CGSr orthogonalization with dense matrices. CGS is known to be very sensitive to machine precision [12], which usually is IEEE-754 compliant. This standard does not define how the machine has to implement floating point operations, but mainly the global behavior and characteristics [10]. On figure 1, we clearly see the difference between 32 bits and 64 bits precision, and the use of an accelerator or not.

Table 3. Precision factor achieved for the dense CGS and CGSr orthogonalization. Factor = Accelerator$_{precision}$ / CPU$_{precision}$

Hardware	Precision	CGS		CGSr	
		Max. Err	Median Err.	Max. Err	Median Err.
Cell	32 bits	170x	22.74x	43x	11x
	64 bits	3x	1.64x	1.07x	0.98x
GPU	32 bits	8x	2.98x	7.1x	1.91x
	64 bits	603x	31.66x	0.9x	0.89x

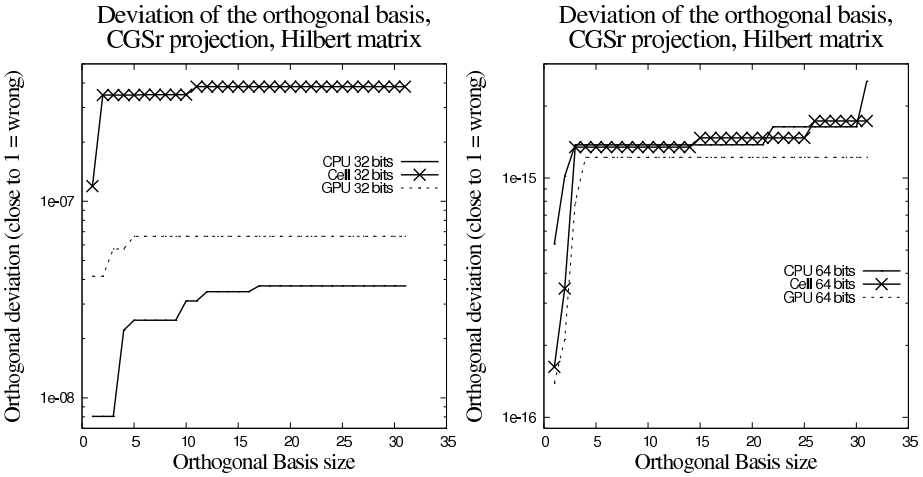

Fig. 2. Precision of the dense CGS process with reorthogonalization on a 10240 square Hilbert matrix, using different hybrid architectures. The left figure shows single precision accuracy results and the right one double precision accuracy results. X axis bears the subspace size, and the Y axis the accuracy of the orthogonal basis, e.g. the worst orthogonality between two vectors of V.

Some statistical indicators were extracted from figures 1 and 2, to try to characterize each device in table 3. As we can see in this table, the Cell is the least accurate device in single precision, and the GPU is less accurate than the CPU. The reason behind is that every computation for the Cell, including division, was done on the non fully IEEE-compliant SPE. Furthermore the Cell is the least IEEE compliant device in SP.

When switching to double precision, then each accelerator tends to be very close to the reference CPU in terms of precision, except for CGS. We see that the GPU is around 600x less accurate at the very end of the CGS orthogonalization process. At this point the basis is not orthogonal anymore even for the reference processor, and so this particular result may not be relevant. The actual GPU error factor varies from 5x to 36x less accurate than the reference processor

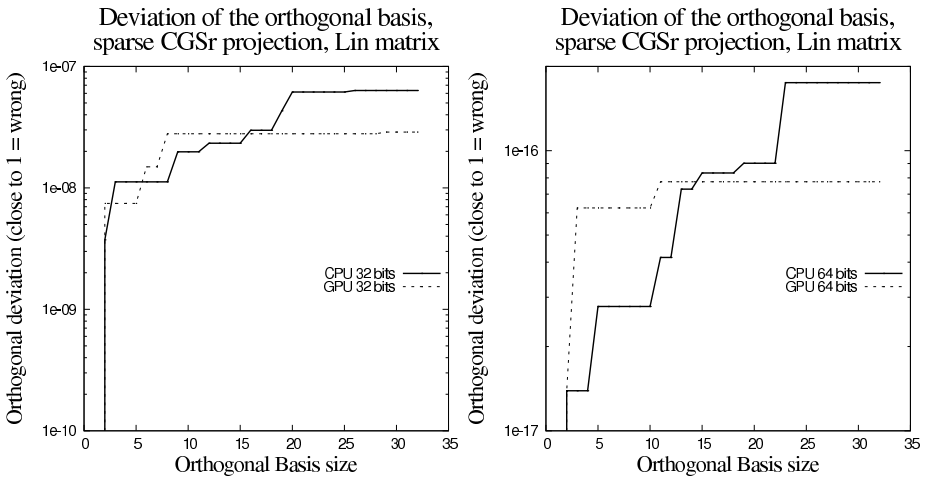

Fig. 3. Precision of the sparse CGS process with reorthognalization on sparse Lin matrix, using different hybrid architectures. Top figure shows single precision results and bottom figure double precision. This matrix is square with a square size of 256k. The total number of non zeros is 1.8m numbers and the filling ratio is 0.0027%. The subspace size starts from 1 and increases along the X axis, and the y axis shows the error of the orthogonal basis, e.g. the worst orthogonality between two vectors of V.

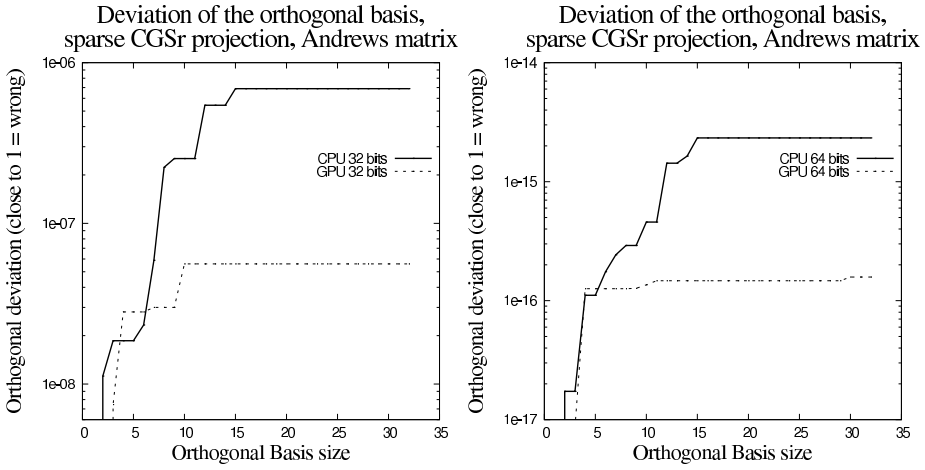

Fig. 4. Precision of the sparse CGSr process on sparse Andrews matrix, using different hybrid architectures. Top figure represents the single precision accuracy results and the lower one double precision.This matrix is square with 60k elements per dimension, and has 760k non zeros and the filling ratio is 0.021%. X axis represents the subspace size and y axis the precision of the orthogonal basis, e.g. the worst orthogonality between two vectors of V.

Table 4. Maximum peak performance for the Gram-Schmidt process in GFlops.

Precision	CPU	GPU	Cell
32 bits	2.667	37	12.8
64 bits	1.333	18	6.4

during the orthogonal basis computation. The interesting thing with the Cell processor is its full double precision IEEE compliance : the result differs from the fully double precision IEEE compliant CPU. It is due to the multi SIMD core nature of the Cell, which will handle computations in a different order than in the CPU case, here used as a single core processor. Some tests conducted on basic dot products parallelized on the Cell provided different results whether we use each core of the Cell as a scalar processor or a SIMD unit. The difference was within the order of the machine precision and it could explain the different Gram-Schmidt orthogonalization behavior.

Sparse matrices We chose two test matrices from the eigenvalues field : Lin and Andrews matrices from the sparse matrices collection of the University of Florida[7]. We experimented the sparse CGSr with the GPU against the CPU. The trends in figure 3 and 4 are different than the dense case. By looking at the matrix H, which is built using dot products of vectors of V, we see very close numerical values for the first columns between processor and GPU. Then some noise appears, which gives 31x higher elements than the CPU-computed result. Because the coefficients of the H matrix are used throughout the orthogonalization process, there is a quick propagation of this noise through each vector operation.
Also, and due to the datasets, the resulting vectors tend to contain numbers relatively close to the machine precision. The GPU truncates these, which explains the apparent better accuracy of the GPU when we seek the "worst" dot product of the V vectors.

5.2 Performance achieved

Expected Results The dominant computing operation is the memory-bandwidth bound matrix vector product, which for a matrix of nz elements does $2nz$ floating operations implying a 2 factor between computation and memory used. Say we have Z GTransfer/s, where the size of one transfer is a double precision element. Then we may do 2Z computations. So the GFLOPs for each architecture will be 2 times the GT/s. From the performance and memory speed shown in table 2 we can deduce the maximum performance expected for these accelerators, summarized in table 4. Table 5 shows the measured performance for each accelerator.

Dense matrices In practice, the observed results differ from the theoretical ones, due to data transfers which cannot utilize the whole available bandwidth, be it for the CPU or the accelerators. The fastest implementation is the GPU's

Table 5. Execution time and performance for the CGSr process for a dense Hilbert matrix of size 10240x10240 and a sparse Lin matrix of size 256kx256k and 1.8m nnz, with a subspace size of 32. Time is expressed in seconds and performance in GFLOPs.

	Dense			Sparse		
	CPU	GPU	Cell	CPU	GPU(CSR)	GPU(Best)
Time(sec)				Time(sec)		
32 bits	5.01	0.33	0.75	1.02	0.156	0.118(DIA)
64 bits	10.12	0.46	1.6	1.8	0.242	0.188(DIA)
Performance(GFLOPs)				Performance(GFLOPs)		
32 bits	1.36	20.5	9.02	1.195	7.82	10.32(DIA)
64 bits	0.69	14.7	4.22	0.65	5.05	6.50(DIA)

one, with a speed-up of 15x in SP and 21x in DP compared to our CPU. The second one is the Cell with a speed-up of 6x compared to the CPU in SP and DP. Table 5 shows the performance results in more details. The efficiency of our solution for the GPU is of 55% in SP and 82% in DP. Concerning the Cell, it is 70% and 66%. So, even with the use of high level BLAS routines, we were able to achieve a decent portion of the accelerators' available bandwidth.

Sparse matrices For the sparse case, we focus on the CUDA-enabled Tesla GPU, and compare it to the CPU. Here, the performance highly depends on the pattern of the matrices : Lin and Andrews in our case. Table 5 shows the actual performance of the sparse CGSr. As we can see, the GPU is the fastest, with a varying factor of 6x to 7x times faster. By using the DIA format, then the performance increases with the GPU, speeding-up by a factor of 9x to 10x. The results with the sparse Andrews matrix are similar, except that the performance is lower for GPU hardware 2.5 GFLOPs in DP and 3 GFlops in SP for the CSR format. Using the Hybrid format from Nvidia, the GPU performs at 3.17 GFLOPs in DP and 4.23 GFLOPs in SP for a speed-up of 5.3x in DP and 3.25x in SP compared to the CPU.

6 Synthesis

We presented implementations of the dense and sparse Classical Gram-Schmidt process with and without reorthogonalization using high level manufacturer BLAS routines for the STI Cell processor and the CUDA-enabled GPUs.

Performance These implementations outperformed their CPU counterpart with a factor of 20x for the GPU and 7x for the Cell processor with dense matrices, and 10x for sparse matrices using the GPU.

Floating point arithmetic The Cell is less IEEE compliant than the GPU in SP and fully IEEE compliant in DP, providing the potential same accuracy as a classical processor. In DP, the GPU is close to be fully IEEE compliant, with differences only in exceptions handling and static rounding modes. Consequently, here is the expected ranking of these architectures in terms of precision :

– $\text{SP}_{accuracy} : \text{Cell}_{acc.} \leq \text{GPU}_{acc.} \leq \text{CPU}_{acc.}$
– $\text{DP}_{accuracy} : \text{GPU}_{acc.} \leq \text{Cell}_{acc.} \leq \text{CPU}_{acc.}$

Precision achieved Concerning the precision with dense matrices, using the Cell implies an actual precision of 10^{-6}. For the GPU, the precision is around 10^{-7}, close to the CPU. In DP, accelerators' results were very close to the CPU ones. We saw varying results in DP due to a different parallel order of execution of the Cell and GPU. For the precision with sparse matrices, the results are less clear, as the GPU seems to give a better orthogonal basis than the CPU, even if it is due to cancellation of some elements of the V basis. Still, it is reliable as the results are close to the reference CPU.

7 Conclusion

The Cell is not IEEE-compliant enough for sensitive SP computation and may only be used mixed with or in full IEEE DP mode if precision is a concern. It provides a good speed-up but at a great programming effort. The GPU has a very good potential because it tends to become more and more IEEE-compliant and it remains backward compatible. Thus, the upcoming Fermi NVidia scientific GPU will fully support the IEEE-norm for both SP and DP, implying a better precision than previous GPUs. It will also integrate L1 and L2 caches improving performances for irregular data patterns like sparse matrices. But because of the parallel nature of these architectures, one has to take into account the parallel execution of the operations, which may have an impact on the precision as with the Cell and the Xeon or maybe the 2 caches of the Fermi. Furthermore, we saw it is possible to accelerate a memory bandwidth bound algorithm such as the Gram-Schmidt process with CUDA-GPUs. Finally, if high-level libraries such as CUBLAS are used then the code may be tested with the next-generation GPUs. These accelerators or new emerging ones may be the key to reach the post-petascale era, certainly mixing precision on levels we know now -32 and 64 bits- but also extended precision : 128 bits. It is thus possible to experiment the same benchmarks to measure the promising improvement in accuracy and performance of the Fermi GPUs as well as new accelerators.

8 Acknowledgements

We thank the CINES[13] research center from Montpellier for providing an access to the IBM QS21 Bladeserver powered by Cell processors as well as an access to Nvidia Tesla servers, and we also thank the IBM research center of Montpelier, which gave us access to a QS22 Double Precision enhanced Cell.

[13] Centre Informatique National de l'Enseignement Supérieur

References

1. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, 2002.
2. Abraham Arevalo, Ricardo M. Matinata, Maharaja (Raj) Pandian, Eitan Peri, Kurtis Ruby, Francois Thomas, and Chris Almond. *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*, chapter 4.6.1 Architecture overview and its impact on programming. IBM, 2008.
3. Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *SC '09: Proceedings of the 2009 ACM/IEEE conference on Supercomputing*, New York, NY, USA, 2009. ACM.
4. T. Braconnier, P. Langlois, and J. C. Rioual. The influence of orthogonality on the arnoldi method. *Linear Algebra and its Applications*, 309(1-3):307 – 323, 2000.
5. Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.
6. NVidia Corporation. Nvidia : Cublas library. Technical report. Whitepaper. Part of CUDA Toolkit.
7. I. S. Duff, Roger G. Grimes, and John G. Lewis. Sparse matrix test problems. *ACM Trans. Math. Softw.*, 15(1):1–14, 1989.
8. Matteo Frigo and Steven G. Johnson. Fftw on the cell processor. www.fftw.org/cell/.
9. Luc Giraud, Julien Langou, Miroslav Rozložník, and Jasper van den Eshof. Rounding error analysis of the classical Gram-Schmidt orthogonalization process. *Numerische Mathematik*, 101(1):87–100, July 2005.
10. David Goldberg. What every computer scientist should know about floating-point arithmetic. In *ACM Computing Surveys*, 1991.
11. Gene H. Golub and Charles F. Van Loan. *Matrix Computations (Johns Hopkins Studies in Mathematical Sciences)*. The Johns Hopkins University Press, Oct. 1996.
12. V. Hernandez, J. E. Roman, and A. Tomas. Parallel arnoldi eigensolvers with enhanced scalability via global communications rearrangement. *Parallel Comput.*, 33(7-8):521–540, 2007.
13. IEEE. IEEE standard for binary floating-point arithmetic. *ACM SIGPLAN Notices*, 22(2):9–25, February 1985.
14. Hans Meuer, Erich Strohmaier, Jack Dongarra, and Host Simon. Architecture share over time. http://www.top500.org/overtime/list/32/archtype.
15. NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.
16. Miroslav Rozložník, Z. Strakos, and Miroslav Tuma. On the role of orthogonality in the gmres method. In *SOFSEM '96: Proceedings of the 23rd Seminar on Current Trends in Theory and Practice of Informatics*, pages 409–416, London, UK, 1996. Springer-Verlag.
17. Yokozawa Takuya, Takahashi Daisuke, Boku Taisuke, and Sato Mitsuhisa. Parallel implementation of classical gram-schmidt orthogonalization using matrix multiplication. *IPSJ SIG Technical Reports*, 2006(63(HPC-106)):31–36(2006), 2006.
18. R. Clint Whaley, Antoine Petitet, and Jack J. Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27:2001, 2001.
19. Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.