An optimized ZGEMM implementation for the Cell BE

Timo Schneider[1], Torsten Hoefler[2], Simon Wunderlich[1], Torsten Mehlan[1], and Wolfgang Rehm[1]

[1]Technical University of Chemnitz, Strasse der Nationen 62,
Dept. of Computer Science, Chemnitz, 09107 GERMANY
{timos, siwu, tome, rehm}@hrz.tu-chemnitz.de

[2]Indiana University, Open Systems Lab,
150 S Woodlawn Ave, Bloomington, IN, 47405 USA
htor@cs.indiana.edu

Abstract:
The architecture of the IBM Cell BE processor represents a new approach for designing CPUs. The fast execution of legacy software has to stand back in order to achieve very high performance for new scientific software. The Cell BE consists of 9 independent cores and represents a new promising architecture for HPC systems. The programmer has to write parallel software that is distributed to the cores and executes subtasks of the program in parallel. The simplified Vector-CPU design achieves higher clock-rates and power efficiency and exhibits predictable behavior. But to exploit the capabilities of this upcoming CPU architecture it is necessary to provide optimized libraries for frequently used algorithms. The Basic Linear Algebra Subprograms (BLAS) provide functions that are crucial for many scientific applications. The routine ZGEMM, which computes a complex matrix–matrix–product, is one of these functions. This article describes strategies to implement the ZGEMM routine on the Cell BE processor. The main goal is achieve highest performance. We compare this optimized ZGEMM implementation with several math libraries on Cell and other modern architectures. Thus we are able to show that our ZGEMM algorithm performs best in comparison to the fastest publicly available ZGEMM and DGEMM implementations for Cell BE and reasonably well in the league of other BLAS implementations.

1 Introduction

Matrix multiplication is used for many standard linear algebra problems, such as inverting matrices, solving systems of linear equations, and finding determinants and eigenvalues [Kny01]. Therefore if a new architecture wants to be successful in the scientific computing environment it is crucial that optimized libraries for problems like matrix multiplication and alike are freely available.

Our initial intent was to port ABINIT, a quantum mechanical ab-inito simulator [GBC+02, GCS+00, BLKZ07], to the Cell BE architecture.[1] ABINIT heavily uses the BLAS [LHKK79, DCHH88] function `ZGEMM` which multiplies two complex matrices and adds

[1]This research is supported by the Center for Advanced Studies (CAS) of the IBM Böblingen Laboratory as part of the NICOLL Project.

them to a third one. All input matrices and scalars are given in double precision. Different groups have already developed optimized matrix multiplication codes for the Cell BE architecture, but those were not meant to be used by other applications but to demonstrate the good single-precision capabilities of the architecture [D.07]. They operate on matrices on a fixed input size, partly use the rather uncommon block data layout for storing the matrices and only work for single precision floating point numbers. Another possibility would have been to use PPC64 optimized BLAS libraries [CGG02, DDE$^+$05] like Atlas [WD98] ot Goto [KR02]. But these libraries do not leverage the potential of the Cell BE completely because they only use the PPC64 core.

Thus, we decided to implement a Cell optimized version of `ZGEMM`. In this paper we will describe the basic algorithm we used as well as the optimization principles we had to apply to get the current result which we will benchmark, too. This paper is organized as follows: Section 2 contains background information on relevant aspects of the Cell Broadband Engine architecture, Section 3 gives an overview of the `ZGEMM` Fortran interface; Section 4 shows how to vectorize and optimize the naive implementation of the used algorithm, Section 5 gives some benchmarking results and tries to explain them, Section 6 draws some conclusions and shows directions for further improvement.

2 Cell Broadband Engine Overview

The Cell BE architecture [CD07] is a multicore microprocessor with one general purpose core called *Power Processing Element* (PPE) and multiple vector co-processing elements, called *Synergistic Processing Elements* (SPE). The PPE is a stripped-down general purpose core to administer the SPEs, which handle the computational workload. It is easy to run conventional software on the PPE due to its compatibility to the PPC64 architecture. The PPE is connected to the system memory and the SPEs via the *Element Interconnect Bus* (EIB), a high-bandwidth circular data bus. Each SPE hosts some local memory, called *Local Store* (LS), an *Synergistic Execution Unit* (SXE) and a Memory Flow Controller (MFC) which connects the SPE to the EIB. The MFC operates independently of the SPU, so memory transactions can be overlapped with computations. Figure 1 gives an overview of the Cell BE architecture.

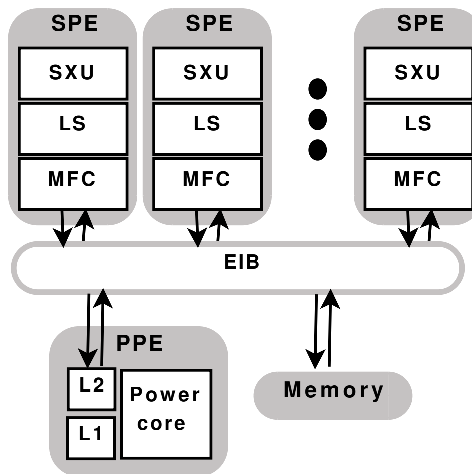

Figure 1: Cell BE architecture

Let's take a closer look at the SPEs now, as the performance of our implementation depends solely on our ability to use their full potential. This is not always easy, as SPEs are special cores: They are meant to fit in the gap between standard desktop PC cores and special number crunching devices like the Graphics Processing Units (GPUs) in graphics

cards. The SPEs can not access the main memory directly, they can only operate on their Local Store which is capable of holding 256 KiB data. One should not think of the LS as of a cache in a standard CPU as it is not updated automatically or transparently to the running process. To get new data into the LS one has to use the Memory Flow Controller to issue a DMA PUT or GET transfer. The boundaries of DMA transfers from and to SPUs have to be 16 byte aligned. DMA transfers yield the best performance when multiples of 128 byte are transferred.

The Synergistic Execution Unit is a vector processor which operates on 128 registers, each 128 bit wide. That means when coping with double precision floating point numbers (which are 8 byte wide) we can do two similar operations simultaneously if we manage to put our input data in a single vector. Unfortunately the Cell BE processors available at the time of writing are very slow when doing double precision arithmetic (1.83 GFlop/s per SPE [WSO$^+$06], which is 14 times lower than the single precision performance). But this should improve with future generations of this chip. The performance cited above can only be reached when fused multiply add instructions are used. These instruction perform the operation $c := a * b + c$ or similar and therefore count as two floating point instructions (FLOP). As all double precision arithmetic instructions need the same number of clock cycles, these instructions yield the best floating point operation per second (Flop/s) ratio.

3 BLAS/ZGEMM

Basic Linear Algebra Subprograms (BLAS) is an widely used application programming interface for libraries to perform basic linear algebra operations such as matrix multiplication. They were first published in 1979 [LHKK79]. Highly optimized implementations of the BLAS interface have been developed by different vendors or groups for many architectures.

`ZGEMM` performs the matrix-matrix operation on input of complex numbers:

$$C := \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$$

Where $op(A)$ specifies if the normal, transposed or conjugated version of the matrix is to be used. A,B and C are matrices consisting of complex numbers and α and β are complex scalars. The Fortran interface is:

```
SUBROUTINE ZGEMM(TRANSA, TRANSB, M, N, K, ALPHA, A, LDA, B,
LDB, BETA, C, LDC)
```

`TRANSA` and `TRANSB` contains the operator to be used on matrix `A` and `B` as a single character which can be `n` (normal), `t` (transposed) or `c` (transposed, conjugated). $op(A)$ is `M` by `K` matrix, $op(B)$ is a `K` by `N` matrix, and `C` is a `M` by `N` matrix. Note that `M`, `N` and `K` refer to the matrices after the operators are applied, not the original input matrices. `ALPHA` and `BETA` correspond to α and β in the equation above. `LDA`, `LDB` and `LDC` specify the first dimension of the input matrices so it is possible to use `ZGEMM` the top-left part of the input matrices only.

The input matrices `A`, `B` and `C` are stored in column major order, as they come from a program written in Fortran. Figure 2 illustrates the meaning of the different `ZGEMM` parameters which deal with the representation of the input matrices.

The tricky part is the operator: Depending on if its normal or not, elements which are stored sequentially in memory can be in one row or one column. As one result element is computed based on one row of $op(A)$ and one column of $op(B)$, we will always have to consider the operators for our memory access.

We investigated works that use Strassen's or Winograd's implementation to reduce the asymptotic complexity of the matrix multiplication [DHSS94]. However, those optimized algorithms work only with well conditioned matrixes which we can not guarantee in the general case. Thus, we chose to implement a traditional $O(N^3)$ algorithm for our ZGEMM.

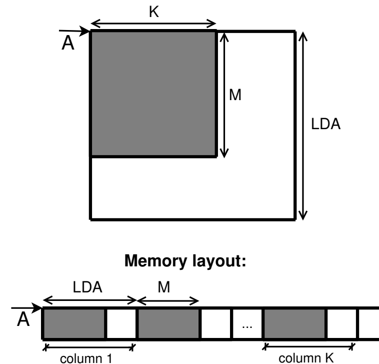

Figure 2: Fortran/ZGEMM Matrix representation

4 Our ZGEMM implementation

We had to apply two important concepts to be able to design a well-performing `ZGEMM` implementation: We partitioned the input data, distributed it among the available SPEs and vectorized all calculations on order to exploit the SIMD architecture.

4.1 Data Partitioning

As the Local Store of an SPE space is limited to 256KiB, the goal should be to save space and memory transfers. A first idea was to load parts of a row of $op(A)$ and a column of $op(B)$ and to compute exactly one element of C. There are some problems with this: depending on the operator, the rows (or columns) of the matrices are stored sequentially in memory or scattered with a displacement (of `LDx`), forcing us to get each element separately. This would decrease performance, as the MFC operates best with memory chunks that are multiples of 128 byte in size.

A better idea is to load blocks instead of lines, and perform small matrix-matrix multiplications instead of scalar products. This gives us independence from the operator: the decision whether rows or columns should be used in the scalar product of the matrix multiplications on the SPEs does not affect performance, as we have random access to the Local Store. Another advantage is the number of operations. For n elements which fit in each input buffer of our Local Store, $\mathcal{O}(n)$ multiply and add operations can be done with the scalar product, but $\mathcal{O}(\sqrt{n^3}) = \mathcal{O}(n^{1.5})$ operations can be achieved with small matrix multiplications. Of course, with more operations on the same amount of local data the total number of memory transfers is reduced.

4.2 Work Assignment

With our partitioning approach, each part of the result matrix can be independently computed with the block row of $op(A)$ and the block column of $op(B)$. The blocks to be

computed are simply distributed circular on the SPEs. Figure 3 illustrates the assignment scheme for 6 SPEs. The shaded result block is computed using the shaded row in $op(A)$ and the shaded column in $op(B)$.

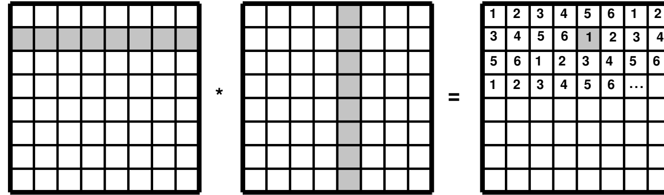

Figure 3: SPE Block assignment

We investigated the use of the PPE with an experimental implementation. The PPE has a theoretical peak performance of 6.4 GFlop/s. Our code spawns N threads on the PPE, each of them computes the same chunk of $op(C)$ as an SPE does[2], using a PPC970 optimized BLAS implementation to perform the computation. Despite the given peak performance of the SPE, we achieved only 1.7 GFlop/s with ATLAS on the PPE, which makes this partitioning scheme suboptimal. Thus, we did not include the PPE measurements in our benchmarks.

4.3 Vectorization

In our matrix multiplication, each element is a 128 bit complex number, consisting of 64 bit double precision floating point values for real part and imaginary part. We can safely assume that only fused multiply add operations are used, as two elements of each matrix are multiplied and added to the temporary scalar product. One multiply-add operation of complex numbers a and b added to y ($y = y + a \cdot b$) is split up like this for its real and imaginary parts:

$$y_{re} := y_{re} + a_{re}b_{re} - a_{im}b_{im}$$
$$y_{im} := y_{im} + a_{re}b_{im} + a_{im}b_{re}$$

This makes 4 fused multiply add operations, with 64 bit operands. With the SIMD-ability of the SPU, two complex multiply-adds can be done instead of one. To use SIMD instructions, the real parts and imaginary parts have to be splitted and packed into separate registers. This can be done with the SPU shuffle instruction. Now the calculation can be done as described above, and the only thing left to do is to separate the real and imaginary part into the result registers before we write back into `C`.
One little obstacle remains: The fused multiply subtract operation on the SPU `spu_msub(a, b, c)` calculates $a \cdot b - c$, but we would need $c - a \cdot b$. To achieve this without adding further instructions to change the sign, the real part can be calculated as follows:

$$y_{re} := a_{re}b_{re} - ((a_{im}b_{im}) - y_{re})$$

In Figure 4 you can see how the blockwise matrix multiplication can be implemented in C, using the SPU intrinsics. [3]

[2] theoretically, $N = 3$ should be optimal

[3] Our code and the tests that were used to obtain the presented benchmark results can be fetched from `http://files.perlplexity.org/zgemm.tar.gz`.

```
#define VPTR "(vector double *)"
vector char high_double = { 0, 1, 2, 3, 4, 5, 6, 7,
                            16,17,18,19,20,21,22,23};
vector char low_double  = { 8, 9,10,11,12,13,14,15,
                            24,25,26,27,28,29,30,31};
vector double rre={0,0}, rim={0,0}, tre, tim, sre, sim;

for (k=0; k < klen; k++, aa += astep, bb += bstep) {
  fim = spu_shuffle(*(VPTR aa), *(VPTR (aa+atstep)), low_double);
  gim = spu_shuffle(*(VPTR bb), *(VPTR bb), low_double);
  fre = spu_shuffle(*(VPTR aa), *(VPTR (aa+atstep)), high_double);
  gre = spu_shuffle(*(VPTR bb), *(VPTR bb), high_double);
  tre= spu_nmsub( fim, gim, sre);
  tim= spu_madd( fre, gim, sim);
  sre= spu_msub( fre, gre, tre);
  sim= spu_madd( fim, gre, tim);
}

rre = spu_shuffle(sre, sim, high_double);
rim = spu_shuffle(sre, sim, low_double);
*(VPTR cc) = spu_add(*(VPTR cc), rre);
*(VPTR (cc+1)) = spu_add(*(VPTR (cc+1)), rim);
```

Figure 4: Inner loop of the blockwise matrix multiplication, implemented in C

5 Benchmarks

This section provides a performance evaluation of our implementation and a qualitative and quantitative comparison to BLAS implementations on other modern architectures.

The current Cell BE chip's SPEs are capable of issuing one double precision arithmetic instruction every six clock cycles. This instruction needs another seven cycles until the result is available in the target register. But if we assume to execute a very large number of data-independent double precision operations we would get a cycles per instruction (CPI) value of 6. Considering FMADD operations and a vector size of two, the theoretical peak performance of a single Cell BE CPU with 8 SPE and a clock

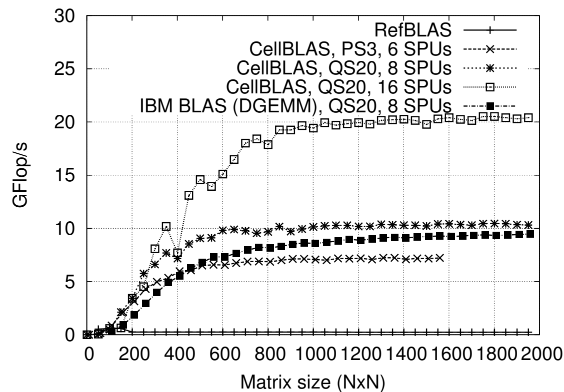

Figure 5: Performance Comparison

rate of 3.2 GHz is

$$R_{peak} = \frac{3.2 * 10^9\,\text{Hz}}{6} \cdot 8\,\text{SPE} \cdot 4\,\text{Flop/SPE} = 17.07\,{}^{\text{GFlop}}/_{\text{s}}$$

This is the number in theory, in practical tests (back to back execution of fused multiply add instructions with no data dependencies) we were able to measure up to 14.5 GFlop/s. This number is said to be the Cell BE double precision peak performance. [WSO+06]

Even though our implementation supports arbitrary matrices, we benchmarked square matrices to enable easy comparisons to other publications. We used `ppu-gcc`, version 4.1.1 with the flags `-O3 -mabi=altivec -maltivec` to compile all PPE code and `spu-gcc`, version 4.1.1 with `-O3` for the SPE code. The Cell BE specific benchmarks were run on a 3.2 GHz IBM QS20 Cell Blade, which contains 2 Cell BE processors with 8 SPEs per processor and two 512 MiB RAM banks and a Playstation 3 running at 3.2 GHz with 200 MiB memory. Both systems run Linux 2.6 (with IBM patches applied).

In our first benchmark (Figure 5), we compare the performance of netlib.org's refblas `ZGEMM` with the IBM `DGEMM` implementation[4] and our optimized implementation for different matrix sizes.

The results show that the our implementation performs very well on Cell BE CPUs. Even though we tried to tune refblas by using different numactl configurations (numactl controls which CPU uses which memory bank), we were not able to achieve more than one Gflop. This is due to the fact that the current compilers do not automatically generate code for the SPUs. Thus, the refblas implementation used only the rather slow PPC core. We outperform the IBM DGEMM implementation by large for all different matrix sizes and our code scales very well to up to 16 SPUs. We can also reproduce similar performance on the specialized Playstation 3 (PS3) hardware (only 6 SPEs are accessible with Linux).

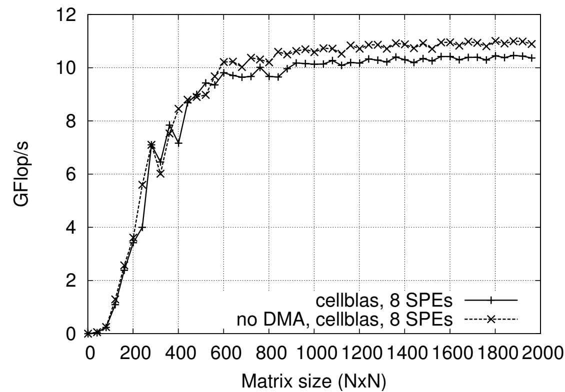

Figure 6: Effects of overlapping the memory accesses with computation

Another optimization technique that has been proposed [CRDI07] is to overlap memory (DMA) accesses with computation. However, this increases the code complexity significantly. To evaluate the potential benefit, we removed all the memory (DMA) accesses from our implementation to simulate the overlap. This invalidates the results but provides an upper bound to the performance-gain due to overlap. Figure 6 shows the comparison to our implementation. Our experiments show that we could gain up to one Gflop/s performance with this overlap technique.

[4]The current IBM BLAS implements no `ZGEMM`. Thus, we used `DGEMM` for comparison, because of its similarity to `ZGEMM`

Our next benchmark compares the Cell BE and our optimized implementation (currently the fastest available) with different modern High Performance Computing (HPC) architectures. We chose a variety of different systems to be able to evaluate the suitability of the Cell BE for scientific calculations using `ZGEMM` as an example. The different systems and their peak floating point performances are described in the following. We leveraged all available processing units (CPUs/Cores) that share a common system memory (are in the same physical node). Thus we compare our multi-core Cell BE implementation with other multi-core BLAS implementations. The test systems are described in the following: a node in Big Red has two dual-core PowerPC 970 MP processors (2.5GHz) with 8GB RAM per node. The peak-performance (with FMADD) is 40 GFlop/s and we ran the IBM ESSL library. We used the Goto BLAS [KR02] library 1.19 on Odin, a dual CPU dual-core Opteron running at 2 GHz with a peak performance of 16 GFlop/s, and Sif, a dual CPU quad-core 1.86 GHz Intel Xeon with 59.5 GFlop/s peak. The theoretically fastest tested system, Jeltz, as two quad-core Intel Xeon 3.0 GHz and a peak performance of 96 GFlop/s. Jeltz runs Mac OS X Tiger and we used the vendor supplied vecLib for our experiments. The absolute performance results for all those systems are plotted in Figure 5.

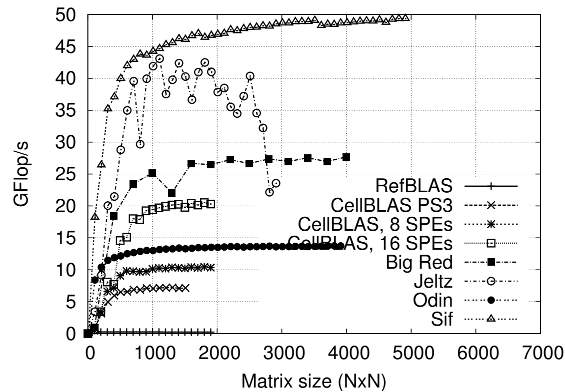

Figure 7: Architectural comparison of ZGEMM Performance

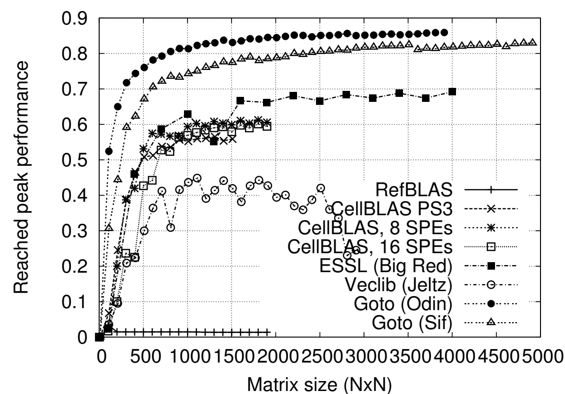

Figure 8: Relative efficiency of different BLAS implementations

Due to memory and CPU time limits, not all matrix sizes could be run on all systems (e.g., the PS3 had only 200 MiB). Our benchmarks show that the current generation Cell BE is not really suited to perform double precision floating point calculations because it is largely outperformed by systems in the same and lower price-range. However, the specialized low-cost Playstation 3 makes a big difference in this price-performance game but its limited memory might be a big obstacle to scientific use.

Those absolute value comparisons do not allow any qualitative comparisons between the different libraries. The main problem is the high variance in peak performance. To compare our implementation to other BLAS libraries, we normalized the measured performance to the peak performance of the architecture to get an estimate of the efficiency of use of the floating point units. We expect a pretty high efficiency on the standard superscalar and cache-based architectures due to the high spatial and temporal locality in matrix multiplication algorithms and decades of development. However, the Cell BE represents a completely new approach of the "explicit cache" (Local Store). Additionally to that, the Cell architecture introduces additional overheads for loading the code to the SPUs. The relative performance results are presented in Figure 7. The highly optimized Goto BLAS implementation delivers the best performance on the available architectures. IBM's Engineering and Scientific Subroutine Library (ESSL) delivers good performance in Power PPC. Our implementation which explores a new CPU architecture is performing very well in comparison to the well established ones and even better than Apple's Veclib.

6 Conclusion and Future Work

Since scientific simulations heavily rely on optimized linear algebra functions we presented in this article an optimized `ZGEMM` implementation for the IBM Cell BE processor. As a part of the BLAS package, the `ZGEMM` routine performs a complex matrix–matrix multiplication. We discussed the strategies to distribute data and to exploit the double precision floating point elements of the SPEs.

The benchmarks showed that the performance of our `ZGEMM` algorithm achieves up to 70% of the peak performance and scales linearly from 1 to 16 SPEs. We assume that our code will also perform well on the next generation Cell BE which supports a fully-pipelined double precision unit that does not stall 6 cycles after every instruction. We compared the algorithm with the IBM `DGEMM` implementation since there is no `ZGEMM` implementation available for Cell. We also showed that even without applying double buffering techniques, the SPEs can be used efficiently under the condition that the number of calculations grow faster with problem size than the access to memory.

Our `ZGEMM` implementation shows the best performance of all publicly available `ZGEMM` or `DGEMM` implementations for Cell BE. Thus, our work may serve as guideline for implementing similar algorithms.

References

[BLKZ07] F. Bottin, S. Leroux, A. Knyazev, and G. Zerah. Large scale parallelized ab initio calculations using a large 3D grid of processors. *submitted to Computational Material Sciences*, 2007.

[CD07] Johns C.R. and Brokenshire D.A. Introduction to the Cell Broadband Engine Architecture. *IBM Journal of Research and Development*, 51:503–519, 2007.

[CGG02] J. Cuenca, D. Gimenez, and J. Gonzalez. Towards the design of an automatically tuned linear algebra library. *Parallel, Distributed and Network-based Processing, 2002. Proceedings. 10th Euromicro Workshop on*, pages 201–208, 2002.

[CRDI07] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementationA performance view. *IBM Journal of Research and Development*, 51:559–572, 2007.

[D.07] Hackenberg D. Fast Matrix Multiplication on Cell (SMP) Systems. Technical report, TU Dresden, Center for Information Services, 2007.

[DCHH88] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of FORTRAN Basic Linear Algebra Subprograms. In *In ACM Trans. Math. Soft., 14 (1988), pp. 1-17*, 1988.

[DDE+05] J. Demmel, J. Dongarra, V. Eijkhout, E. Fuentes, A. Petitet, R. Vuduc, R.C. Whaley, and K. Yelick. Self-adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93(2):293–312, Feb 2005.

[DHSS94] Craig C. Douglas, Michael Heroux, Gordon Slishman, and Roger M. Smith. GEMMW: a portable level 3 BLAS Winograd variant of Strassen's matrix-matrix multiply algorithm. *J. Comput. Phys.*, 110(1):1–10, 1994.

[GBC+02] X. Gonze, J.-M. Beuken, R. Caracas, F. Detraux, M. Fuchs, G.-M. Rignanese, L. Sindic, M. Verstraete, G. Zerah, F. Jollet, M. Torrent, A. Roy, M. Mikami, Ph. Ghosez, J.-Y. Raty, and D.C. Allan. First-principles computation of material properties: the ABINIT software project. *Computational Materials Science*, 25:478–493, 2002.

[GCS+00] X. Gonze, R. Caracas, P. Sonnet, F. Detraux, Ph. Ghosez, I. Noiret, and J. Schamps. First-principles study of crystals exhibiting an incommensurate phase transition. *AIP Conference Proceedings*, 535:163–173, 2000.

[Kny01] Andrew V. Knyazev. Toward the Optimal Preconditioned Eigensolver: Locally Optimal Block Preconditioned Conjugate Gradient Method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.

[KR02] Goto K. and Geijn R. On reducing TLB misses in matrix multiplication. Technical report tr-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002.

[LHKK79] C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. In *In ACM Trans. Math. Soft., 5 (1979), pp. 308-323*, 1979.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[WSO+06] Samuel Williams, John Shalf, Leonid Oliker, Shoaib Kamil, Parry Husbands, and Katherine Yelick. The potential of the cell processor for scientific computing. In *CF '06: Proceedings of the 3rd conference on Computing frontiers*, pages 9–20, New York, NY, USA, 2006. ACM.