International Journal of Advanced Science and Technology
Vol. 27, February, 2011

Acceleration of Correlation Matrix on Heterogeneous Multi-Core CELL-BE Platform

Manish Kumar Jaiswal
Assistant Professor, Faculty of Science & Technology,
The ICFAI University, Dehradun, India.
manish.iitm@yahoo.co.in

Abstract

A parallel implementation of the highly useful and computational intensive correlation matrix arithmetic is presented here. A heterogeneous multi-core CELL-BE processor platform has been used for the purpose of accelerating the performance of the kernel. An extensive measures of the performance reflects the massive capability of the CELL-BE processor architecture. In comparison with a Power Processor, the CELL-BE achieves about 500 time better performance, whereas it is showing around 120 times better performance with that of a dual-core Pentium-4 processor.

Keywords: *CELL-BE, Multi-Core, SPU, PPU, Threading, SSTA, Correlation, Covariance, Mean, Statistical Analysis.*

1 Introduction

Matrix arithmetic are vital part of a large set of applications, including scientific computing, financial & marketing, CAD tools, signal processing, future wireless communication (software define radio), computational physics (modeling 2-D structure), computational chemistry (design and analysis of molecular structure), etc [1, 2, 3, 4, 5, 6, 7, 8]. They plays a dominating role in almost all of the application where they are in use. Specially, for the larger data size all the applications degrades in their performance very poorly. So, there is great need for the acceleration of the performance of the critical matrix arithmetic used in these applications. In present work we have focused on financial & marketing domain matrix arithmetic, also a vital part of the Statistical Timing Ananlysis of EDA tools for Electronics circuits.

Marketing & financial economics has several applications [4, 5, 6]. Some of these are, risk management, option pricing, and to forecast demand for a group of products in order to realize savings by properly managing inventories [4, 6], etc. These application are heavily depends on the correlation matrix computation [4, 5, 6]. A correlation is a number that describes the degree of relationship between two variables. The uses of correlation matrix computation in this domain can be seen from [6].

A lot of work have been reported for the acceleration purpose of the computation intensive kernels. There are two approaches has been utilized for the purpose. One most obvious is exploration of parallelism on multi-core / multi-processor platform and other is parallel implementation on a dedicated hardware platform. Here, a heterogeneous multi-core platform, CELL-BE processor [9] has been used as a target platform for the acceleration purpose of the correlation matrix computation.

The correlation is one of the most common and useful statistical analysis tool used in financial modeling and analysis [4, 5, 6]. Several software packages are available, but most dominating one for this field is by R-Project [10]. R-project is a software package for statis-

tical computation on single-core platform. Thus, the aim of the work is going to enhance the performance of the correlation matrix on CELL-BE processor.

2 Background

This section includes the brief overview of the topics we are presenting here. It contains the description of the Correlation Matrix and CELL-BE Processor.

2.1 Correlation Matrix

This section consists of the Correlation matrix mathematics. This includes computation of Mean, Variance, Covariance Matrix and Correlation Matrix form a set Random Variables. Let us suppose we have a Random Variables X with N outcomes. Then we can define,

MEAN: The mean of a discrete random variable X is a weighted average of the possible values that the random variable can take. The mean of each of random variables can be given as,

$$\mu = \frac{\sum X_i}{N} \tag{1}$$

VARIANCE: The variance of a discrete random variable X measures the spread, or variability, of the distribution. The variance of each random variable can be given as,

$$\sigma^2 = \frac{\sum (X_i - \mu)^2}{N} \tag{2}$$

Now let us suppose we have a set of M random variables with N outcomes, then we will have a array of MEAN, μ of size M . In this case we will have COVARIANCE MATRIX which will measure the spread of the distribution. It will form a matrix of size MxM.

COVARIANCE MATRIX: The element of the covariance matrix can be given as,

$$\sigma_{ij}^2 = \frac{\sum (X_i - \mu_i)(X_j - \mu_j)}{N} \tag{3}$$

STANDARD DEVIATION ARRAY: The element of the standard deviation array can be given as,

$$\sigma_{ij} = (\sigma_{ij}^2)^{1/2} \tag{4}$$

CORRELATION MATRIX: Correlation is a statistical technique that can show whether and how strongly pairs of variables are related. The elements of the correlation matrix can be given as,

$$\rho_{ij} = \frac{\sigma_{ij}^2}{\sigma_i \sigma_j} \tag{5}$$

2.2 CELL-BE Processor

The Cell concept was originally thought up by Sony Computer Entertainment inc. of Japan, for the PlayStation 3. The genesis of the idea was in 1999 when Sonys Ken Kutaragi "Father of the PlayStation" was thinking about a computer which acted like Cells in a biological system [11].

The Cell Architecture was developed jointly by the Sony, Toshiba and IBM to provide power-efficient and cost-effective high performance processing for a wide range of applications spanning areas like Cryptography, Graphics transform and lighting, Physics, Fast-Fourier transforms (FFT), matrix operations and many more. The Cell Architecture is as shown in Fig. 1

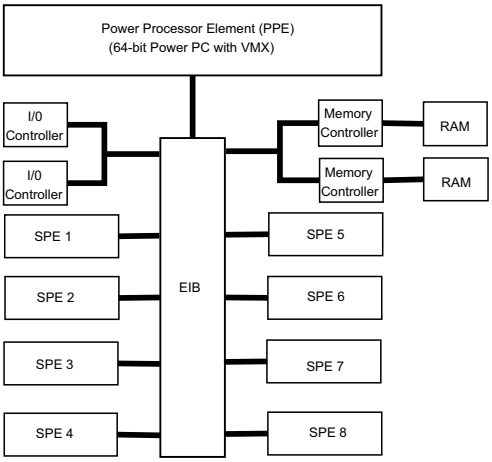

Figure 1: CELL Architecture[1]

An individual hardware Cell is consists of a number of elements:

- 1 Power Processor Element (PPE).
- 8 Synergistic Processor Elements (SPEs).
- Element Interconnect Bus (EIB).
- Direct Memory Access Controller (DMAC).
- 2 Rambus XDR memory controllers.
- Rambus FlexIO (Input / Output) interface.
- Capable of running at speeds beyond 4 GHz.
- Memory bandwidth: 25.6 GBytes per second.
- I/O bandwidth: 76.8 GBytes per second.
- 256 GFLOPS (Single precision at 4 GHz.
- 256 GOPS (Integer at 4 GHz).
- 25 GFLOPS (Double precision at 4 GHz).
- 235 square mm.
- 235 million transistors.

Power consumption has been estimated at 60 - 80 Watts at 4 GHz

The PPE is a conventional microprocessor core which sets up tasks for the SPEs to do. In a Cell based system the PPE will run the operating system and most of the applications but the compute intensive parts of the OS and applications will be offloaded to the SPEs. The PPE is a 64 bit, “Power Architecture” processor with 32K L1 Instruction, 32K L1 Data and 512K L2 cache memory. The PPE is capable of running POWER or PowerPC binaries. The PPE

[1]http://www.blachford.info/computer/Cell/Cell1_v2.html

is a dual issue, dual threaded, in-order processor. Architecture is an old style RISC design. It includes support for 4 (single precision) floating point units capable of 32 GFLOPS and 4 Integer units capable of 32 GOPS (Billions of integer operations per Second) at 4GHz. The SPEs also include a small 256 Kilobyte local store (LS) instead of a cache. According to IBM a single SPE (which is just 15 square millimeters and consumes less than 5 Watts at 4GHz) can perform as well as a top end (single core) desktop CPU given the right task. The SPEs do however have 128 registers each of 128-bits and this gives plenty of room for the compiler to unroll loops and use other software techniques. Each SPE is capable of 4 X 32 bit operations per cycle (8 if you count multiply-adds). In order to take full advantage of the SPEs, the programs running will need to be "vectorized".

One way in which SPEs operate differently from conventional CPUs is that they lack a cache and instead use a Local Store 256 Kbytes per SPE. Local stores are like cache in that they are an on-chip memory but the way they are constructed and act is completely different. They are supported with a EIB, a communication bus, internal to the Cell processor which connects the various on-chip system elements: the PPE processor, the memory controller (MIC), the eight SPE co-processors, and two off-chip I/O interfaces, for a total of 12 participants. The EIB also includes an arbitration unit which functions as a set of traffic lights. The EIB consists of 4 x 16 byte rings which run at half the CPU clock speed and can allow up to 3 simultaneous transfers. At maximum concurrency, with three active transactions on each of the four rings, the peak instantaneous EIB bandwidth is 96B per clock (12 concurrent transactions * 16 bytes wide / 2 system clocks per transfer).

While this figure is often quoted in IBM literature it is unrealistic to simply scale this number by processor clock speed. According to IBM only about two thirds of this is likely to be achieved in practice.

Cell contains a dual channel next-generation Rambus XIO macro which interfaces to Rambus XDR memory. The memory interface controller (MIC) is separate from the XIO macro and is designed by IBM. The XIO-XDR link runs at 3.2 Gbit/s per pin. Two 32-bit channels can provide a theoretical maximum of 25.6 GB/s. FlexIO and XDR RAM both have a technology called "FlexPhase" which allow signals to come in at different times reducing the need for the wires to be exactly the same length, this will make life considerably easier for board designers working with the Cell. Both SPEs & PPE access memory in blocks of 128 bytes (one Cache line).

The more details regarding Cell processor can be obtain from [12].

2.3 Programming the Cell Processor

The programming on the Cell Broadband Engine (Cell BE) processor is done using the Cell SDK. The SDK is composed of development tool chains, software libraries and sample source files, a system simulator and a Linux kernel which together support the capabilities of Cell BE [13]. For more information on programming the Cell BE, refer [13].

The PPE runs all the Power PC applications and the Operating System. It is also responsible for thread handling and resource management among the SPEs. The PPEs Linux kernel is responsible for scheduling the SPEs execution, run-time loading, passing parameters to the SPE and notification of events and errors. Additionally it also manages the virtual memory, including mapping of SPEs LS and Problem State (PS) to effective address space.

2.3.1 SPE Programming

The SPU executes both single-precision and double-precision floating-point operations. It operates basically on SIMD (Single Instruction Multiple Data) vector operands, both fixed-point and floating-point those are 4 bytes long. It supports big-endian data ordering and 16-byte operand accesses between storage and 128-bit vector registers. It supports the following data types: byte 8-bits, halfword 16-bits, word 32-bits, doubleword 64-bits, and quadword 128-bits.

The SPU has two pipelines: even (pipeline 0) and odd (pipeline 1). The SPU can issue and complete up to two instructions per cycle, one in each of the pipelines. A dual-issue is said to occur when the fetch group has two issue-able instructions in which the first instruction can be executed on the even pipeline and the other instruction can be executed on the odd pipeline. The higher the dual issue rate, higher is the performance. Refer [12] Appendix B for pipeline classification of SPU instructions.

The SDK provides a minimal set of basic intrinsics and built-ins that make the underlying architecture (ISA) and SPE hardware accessible from C programming language. There are three classes of intrinsics:
Specific Intrinsics: Here there is a one-to-one mapping with a single assembly- language instruction. They are prefixed by the string si_
Generic Intrinsics: They map to one or more assembly-language instructions as a function of the type of input parameters. They are prefixed by the string spu_
Composite Intrinsics: Constructed from a sequence of specific and generic intrinsics. They are prefixed by the string spu_

The detailed description of the spu intrinsics can be obtained from [14].

3 Design Strategy on CELL-BE

This section include the design details of Correlation matrix on CELL processor. First we will discuss the programming strategy on CELL processor in brief. As CELL-BE has two types of processors, PPU & SPU, we need two sets of program for any implementation. One will be dedicated to PPU and acts as top controller and other talks about functioning of SPU. There may be different sets of code for all SPU's. General thumb rule for any Cell design set PPU as a control unit, which used to handle the SPU threads (context creation, launching threads, dealing with synchronization in between the SPU threads, destroying the context, etc), control block creation to provide the require information to SPU's. `Control Block` is structure, which used to handle common information, needed by both PPU & SPU's. It is associated with each SPU thread while thread creation by PPU.

Implementation on Cell has several issue to deal with while programming. These are as follows:

3.1 Design Issue in Cell Implementation

Exploration of parallelism in algorithm: This step require the investigation of algorithm to be implemented. We need to extract sequential and parallel part of the algorithm. And decide where (PPU/SPU)to implement them.

Divide algorithm in several smaller part, so that it will ease to implement.

Assigning various task to PPU and SPU's: Decide to separate the algorithm between the PPU and SPU, also among SPU's. Usually PPU prefers to handle control part and SPU's for massive arithmetic computations.

Data partition and handling data transfer between main memory and SPU LS: SPU's can't work directly on main memory contents. And they need data to be worked on, in their local store. And since SPU has limited local store memory (256 KB), for larger data size, it should be partitioned in chunks of the data to transfer in-between main memory and SPU LS. Also these chunks should be handled efficiently (in multi buffering fashion) to reduce the memory access overhead.

Data Alignment: As transfer of the data between main memory and SPU LS happened in the 16-bytes aligned format, input data must first be aligned before computation starts.

Writing SPU code (kernel code) in SIMD format: Since SPU are SIMD processor code to run on SPU should be in SIMD format to achieve efficiency of SPU's.

SPU Code size & data size: Taking care of code size and data size on SPU LS: Due to limited size of SPU LS, and the requirement of putting both code segment and data segment on it to be work by the SPU, their size should be well balanced.

Synchronization: Synchronization is required between SPU's and PPU, if further computation dependent on the previous computation results, or some different spu code has to run after some old one.

4 Implementation

Taking above mentioned point in to account we have implemented the Correlation matrix in four steps. The computation proceeds in the following sequence:

- Mean → Covariance Matrix → Standard deviation → Correlation Matrix.

The flowchart for the entire computation is shown in the Fig. 2.

Now we will discuss the complete implementation details. Let us suppose that we have a set of random variables M each with N outcomes. This will form a matrix of size $M \times N$, says `ran_arr`, will be the input matrix. Now the first step will be data alignment.

4.1 Data Alignment

As we need data in the row by row format for entire computation of mean we have aligned it both way, the row and the column alignment of the input matrix. Let it be NEW_M & NEW_N.

Now implementation proceeds in three steps. First step include computation of `mean_arr` and an intermediate matrix `new_ran_arr` (used for covariance matrix computation) by a spu program for mean computation (`spu_mean.c`). Next Step will compute the covariance matrix on `new_ran_arr`, which will followed by the standard deviation and correlation matrix computation in third step. They will proceed as follows:

4.2 Mean Computation

4.2.1 PPU SIDE

- Update `Control_Block`, includes passing memory addresses, data sizes, and other parameters.
- Create Context, load `spu_mean` spu_program for mean computation, launch threads.
- Wait for SPU to finish the mean computations.
- Join the threads.

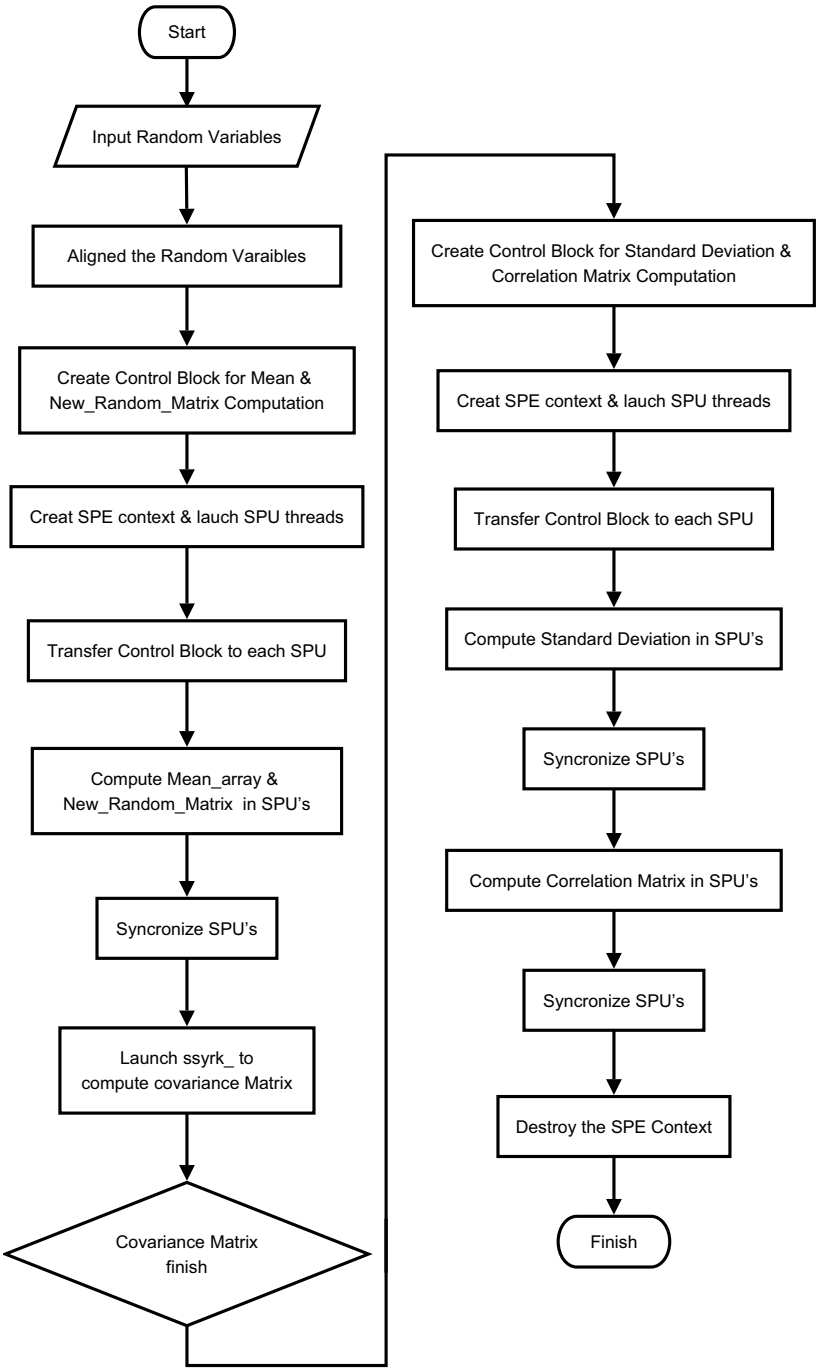

Figure 2: Flowchart for Cell Implementation

4.2.2 SPU SIDE

- Read the `Control_Block`.
- Fetch the required memory location, data size and parameters from `Control_Block`.
- Single buffering has been used for memory transfer from main memory to local store and in opposite direction. In a single transfer maximum of 4096×4 byte (16 MB) can

be transfer. Depending on the size of `ROWS & COLUMNS` buffering has been implemented. It is as follows:

- If the `ROW` size is greater tha n 4096×4 byte, computation has been done by row by row order. That is data has been transfer for a single row in chunks to compute a mean of a row.
- If `ROW` size is less that $4096\ times4$ byte then many row can be transferred and mean for each has been computed.
- The data for mean from `mean_arr` has been also transferred in chunks of 4096×4 bytes.

- Along with the computation of mean, a new matrix has also been created using input matrix and corresponding means, which is latter used for the computation of covariance matrix `covar_mat`. This is obtained by subtracting the corresponding mean of a row with all the element of that row of the input matrix. Thus it will be a matrix of size $NEW\_M \times NEW\_N$, says `new_ran_arr`.

- This `new_ran_arr` data also has been transferred to main memory as above depending on row and column size. Thus in this part we have mean and new random matrix computation.

4.3 Covariance Matrix Computation

This part used the `new_ran_arr` matrix created in first part to generate the covariance matrix `covar_mat` of size $NEW\_M \times NEW\_M$. This is done as follows:

- The math behind `covar_mat` is to take average over sum of product of all subsequent element of e ach pair of rows.

- This is simply can be obtained by a matrix multiplication of a matrix to its transpose matrix. For this there is a well developed blas routine `ssyrk_` is available. It is used for computing

$$C = \alpha AA^* + \beta C$$

where, C is output matrix, A is input matrix, A^* is transpose of A, α and β are constant values.

It is used as follows:

$$ssyrk\_(\text{“}L\text{”},\ \text{“}T\text{”},\ \&NEW\_M,\ \&N,\ \&\alpha,$$
$$new\_ran\_arr, \&NEW\_N, \&\beta, covar\_mat, \&NEW\_M\ );$$

where,

NEW_M & NEW_N is number of rows and columns of `new_ran_arr`,

N is initial size of matrix.

Value of α is one and that of β is zero.

`covar_mat` is output covariance matrix of size $NEW\_M \times NEW\_M$.

Thus we can get `covar_mat` easily and efficiently by using `ssyrk_` routine.

4.4 Correlation Matrix Computation

Third step consist of obtaining standard deviation array and correlation matrix form covariance matrix. This is computed as follows:

4.4.1 PPU SIDE

- Update the `Control_Block` with new informations.
- Load the program `spu_corr` a spu program used for computing standard deviation and correlation matrix.
- Create thread.
- Wait for all SPUs for finishing standard deviation computation using `spu_mailbox` to synchronize all SPUs for further step of computing the correlation matrix. After getting message from all SPUs, send signal to all SPUs to proceed further.
- Wait for all SPUs to finish correlation matrix computation.
- Join the threads.
- Destroy the SPU context.
- After that depending on the verifying condition we can verify the result obtained from SPUs with newly generated results using only PPU.

4.4.2 SPU SIDE

- Read the `Control_Block`.
- Fetch the required memory location, data size and parameters from `Control_Block`.
- Here also single buffering has been used to transfer data to and fro between main memory and SPU local stores.
- First, standard deviation has been computed on the diagonal elements of `covar_mat` from main memory and by taking square-root of them standard deviation array, namely `std_arr`, has formed.
- After finishing standard deviation computation synchronize all the SPU using spu mailbox and PPU.
- Perform the correlation matrix computation using `std_arr` and `covar_mat` datas. The math behind it is mentioned in the chapter 2.
- Here also single buffering has been used depending on the value of NEW_M, as in the case of first part.

Thus the entire flow completed with this part, and we will finally have a array of MEAN, `mean_arr` of size NEW_M; a COVARIANCE Matrix,`covar_mat` of size $NEW\_M \times NEW\_M$; a STANDARD DEVIATION Array, `std_arr` of size NEW_M and the CORRELATION Matrix, `corr_mat` of size $NEW\_M \times NEW\_M$.

5 FLOP Performance Estimation

Here we will discuss the number of floating point operation processed in the entire computation. In the implementation on the Cell processor, we need to aligned the input data size to work on them by SPU. Thus the actual size of the data modified. And this would change the actual number of operation form the required one. So, here we will discuss both, the required one and the actual number of FLOP.

5.1 MEAN

5.1.1 Required

Let input matrix dimension is $M \times N$. Output dimension will be M. For one element of Output we need $N - 1$ addition, 1 reciprocation and 1 multiplication. Total $N + 1$ operations for one output element. For M element it will be $M(N + 1)$ floating point operation.

5.1.2 Actual

Let input matrix dimension is $NEW\_M \times NEW\_N$. Output dimension will be NEW_M. For one element of Output we need $NEW\_N - 1$ addition, 1 reciprocation and 1 multiplication. Total $NEW\_N + 1$ operations for one output element. For M element it will be $NEW\_M(NEW\_N + 1)$ floating point operation.

5.2 New Matrix (`new_ran_arr`)

5.2.1 Required

Let input matrix dimension is $M \times N$. Output dimension will be $M \times N$. Total operation will be $M * N$ subtraction.

5.2.2 Actual

Let input matrix dimension is $NEW\_M \times NEW\_N$. Output dimension will be $NEW\_M \times NEW\_N$. Total operation will be $NEW\_M * NEW\_N$ subtraction.

5.3 COVARIANCE MATRIX (ssyrk_)

Here, as per ssyrk_ description, total operation will be $N * M * (M + 1)$.

5.4 STANDARD DEVIATION

5.4.1 Required

Input size is M, Output size M. Total operation will be M square-root.

5.4.2 Actual

Input size is NEW_M, Output size NEW_M. Total operation is NEW_M square-root.

5.5 CORRELATION MATRIX

5.5.1 Required

Input size is $M \times M$, Output size $M \times M$ symmetric matrix. For one output we need two multiplications and one reciprocation. Total operation will $3 * M * (M + 1)/2$.

5.5.2 Actual

Input size is $NEW\_M \times NEW\_M$, Output size $NEW\_M \times NEW\_M$. For one output we need two multiplications and one reciprocation. Total operation will be $3 * NEW\_M * NEW\_M$.

Thus total requires FLOP will be as follows:

$$\begin{aligned} FLOP\_Req &= M * (N + 1) + M * N + N * M * (M + 1) \\ &\quad + M + 3 * M * (M + 1)/2 \\ &= 3 * M * N + 7 * M/2 + N * M2 \\ &\quad + 3 * M2/2 \end{aligned} \tag{6}$$

and, the actual FLOP will be given as,

$$\begin{aligned} FLOP\_Act &= NEW\_M * (NEW\_N + 1) \\ &\quad + NEW\_M * NEW\_N \\ &\quad + N * M * (M + 1) + NEW\_M \\ &\quad + 3 * NEW\_M * NEW\_M \\ &= 2 * NEW\_M * NEW\_N + 2 * NEW\_M \\ &\quad + 3 * NEW\_M2 + N * M2 + N * M \end{aligned} \tag{7}$$

6 Performance Result and Comparison

We have taken a extensive performance measure of the developed kernel. A long range of matrix size has been utilized for getting the better understanding of the performance. The performance of CELL-BE with varying number of SPU's has been evaluated and have been compared with that of a single PPU unit as well as with Standalone Intel Pentium-4 Dual-Core processor (2-GB RAM, 3-GHz freq.) with various optimization mode.

The various aspects of performance measure have been shown in figures, Fig.[3, 4, 5, 6].

From the performance results we can conclude that CELL is giving much better performance than a PPU unit. For smaller data size PPU is performing well than CELL. This is because of overhead of creating & launching SPU threads, which is relatively less significant for larger data size.

Also as we are increasing the number of SPUs the performance are boosting up much. But for smaller data size less SPU's are beneficial. This happens because of memory transfer overhead between main memory and LS. Since memory transfer happens only in block format, for smaller data size there will be more blocks have to be transfer, which dominates the computation time for smaller data.

With entire SPE's the performance of CELL-BE is approximately 500 time better than a Power Processor, whereas it capable to getting more than 120 time better than Dual-Core Intel Pentum-4.

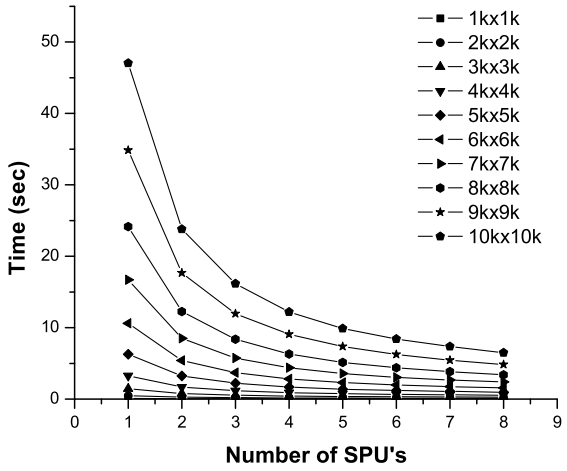

Figure 3: Time taken (sec) by CELL-BE with increasing Data Size & Number of SPU's

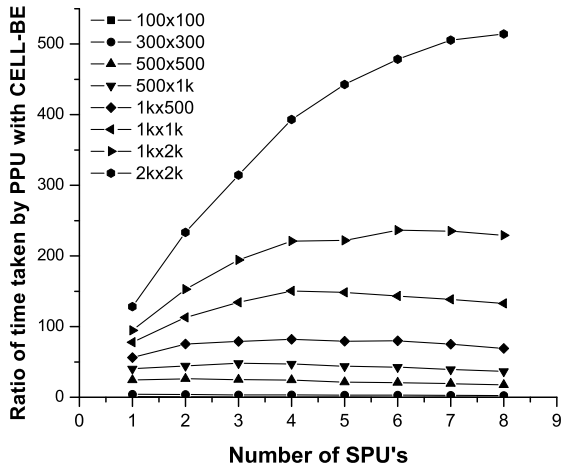

Figure 4: Ratio of time taken by PPU with Time taken by CELL-BE

7 Conclusion

The presented work has shown a implementation of Correlation Matrix on a heterogeneous platform, CELL-BE Multi-Core processor. The requires a well partitioning of algorithm and input as well as intermediate data. Careful design of parallel SPU code with efficient memory access pattern leads to a tremendous performance improvement compared to a workstation. And, thus CELL-BE is proved to a lightening platform for the high performance computation. From our experiments, we have feel the several improvement can be adhere in with some pro-cons. Like, `Multi-Buffering` can be used. But it has been seen from experiments if we are using maximum number of SPU, impact of multi-buffering is very insignificant. The reason behind this is, if we use all SPU's, all the channels on EIB will be used for communication between main memory and SPU's LS. And thus there will not be any extra slot which can be exploit using multi-buffering. But in the case of less SPU's, some slots will be available, that can be used in multi-buffering. Also, multi-buffering will increase the size of code on SPU local store. So, we have to take care of balance between code size and data

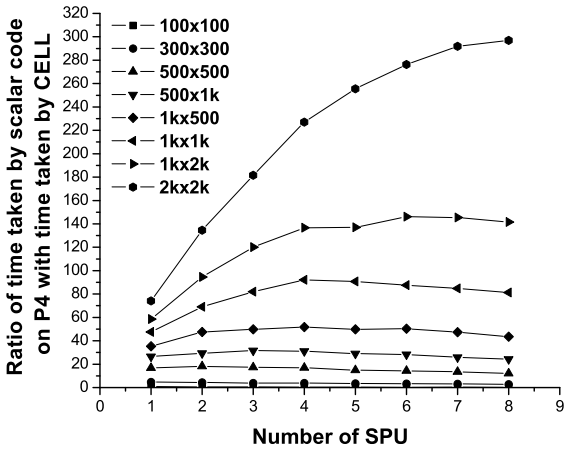

Figure 5: Ratio of time taken by Intel P4 Dual Core 3GHz, 2GB RAM (-O2 optimization) with Time taken by Complete CELL-BE

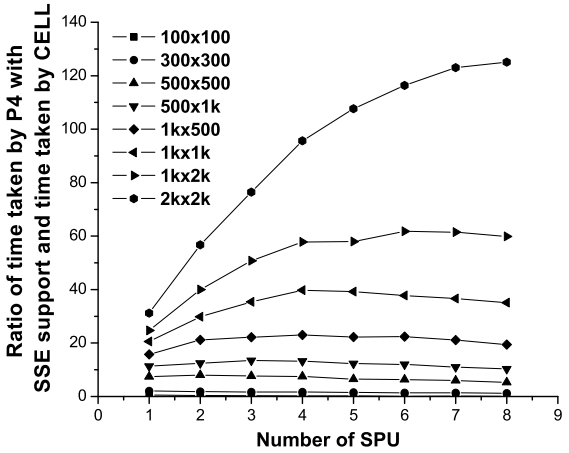

Figure 6: Ratio of time taken by Intel P4 Dual Core 3GHz, 2GB RAM (with **sse** instruction set) with Time taken by Complete CELL-BE

size on SPU local store. Another improvement could be `Loop-unrolling`. This will also increase the code-size on SPU local store. This inclusion will improve the performance to a good level. The one more could be `Handling of unaligned data`. Above we have assumed that input data address is aligned. Taking care of unaligned data will not affect the performance (very insignificant effect).

8 Acknowledgment

The authors would like to thanks IBM India Pvt. Ltd. Bangalore, INDIA for providing the opportunity to work on the heterogeneous CELL-BE Multi-Core processor. I would like to thanks my project manager Mr. Bhavesh Buddhabhatti, and my mentor Mr. Chakrapani, along with all the team members of CELL-BE software development team for their continuous support and help in understanding the CELL-BE architecture and its programming details.

References

[1] A. Edelman, "Large dense numerical linear algebra in 1993: The parallel computing influence," *International Journal Supercomputer Applications*, vol. 7, pp. 113–128, 1993.

[2] R. Harrington, "Origin and development of the method of moments for field computation," *IEEE Antennas and Propagation Magazine*, vol. 32, no. 3, pp. 31–35, Jun 1990.

[3] J. L. Hess, "Panel Methods in Computational Fluid Dynamics," *Annual Review of Fluid Mechanics*, vol. 22, pp. 225–274, Jan 1990.

[4] S. K. Mishra, "Completing Correlation Matrices of Arbitrary Order by Differential Evolution Method of Global Optimization: A Fortran Program," March 2007. [Online]. Available: SSRN: http://ssrn.com/abstract=968373

[5] S. K. Mishra, "The Nearest Correlation Matrix Problem: Solution by Differential Evolution Method of Global Optimization," April 2007. [Online]. Available: SSRN: http://ssrn.com/abstract=980403

[6] M. Lawrence, "Statistics: The Covariance and Correlation Matrix." [Online]. Available: http://investing.calsci.com/statistics4.html

[7] G. Govindu, S. Choi, V. Prasanna, V. Daga, S. Gangadharpalli, and V. Sridhar, "A high-performance and energy-efficient architecture for floating-point based LU decomposition on FPGAs," in *Proceedings of 18th International Parallel and Distributed Processing Symposium, 2004.*, April 2004, pp. 149.

[8] X. Wang and S. Ziavras, "Parallel direct solution of linear equations on FPGA-based machines," in *Proceedings of International Parallel and Distributed Processing Symposium, 2003*, April 2003, pp. 8.

[9] "Multicore Acceleration." [Online]. Available: http://www-128.ibm.com/developerworks/power

[10] "R-Project." [Online]. Available: http://www.r-project.org/

[11] "Cell Processor." [Online]. Available: http://www.blachford.info/computer/Cell/

[12] "IBM Cell Broadband Engine Programming Handbook, Version 1.1," April 2007.

[13] "IBM Software development kit for Multi-core Acceleration Version 3.0, Programmer's guide."

[14] "IBM C/C++ Language Extensions for Cell Broadband Engine Architecture, Version 2.4."