CS 107 – Computer Architecture

Comparing the performance of a CPU vs. a GPU

Nimit Dhulekar
nimit.s.dhulekar@dartmouth.edu

Jyotishman Nag
jyotishman.nag@dartmouth.edu

Table of contents:

1. Abstract: 3
2. Introduction: 3
   2.1 PlayStation 3 Architecture and Configuration 3
      2.1.1 Power Processor Element (PPE)- 4
      2.1.2 Synergistic Processor Elements (SPEs)- 4
      2.1.3 EIB and DMAC- 6
      2.1.4 Programming model- 6
      2.1.5 The GNU Compiler for Java 6
   2.2 PC Configuration 6
3. Background: 6
   3.1 Matrix multiplication 7
      3.1.1 Naïve Matrix Multiplication 7
      3.1.2 (j,i,k) JAMA loop ordering 7
      3.1.3 (i,k,j) Pure row-oriented loop order 8
   3.2 Matrix Inverse 8
   3.3 Matrix LU Decomposition 9
   3.4 Matrix QR Decomposition 9
   3.5 Singular value decomposition 10
   3.6 Eigen value decomposition 10
   3.7 Sorting Algorithms 10
      3.7.1 Merge Sort 10
      3.7.2 Quicksort 11
      3.7.3 Radix Sort 12
4. Results: 13
5. Conclusions: 20
6. References: 20
7. Appendix 21

1. Abstract:

The goal of the project is to investigate and study the performance of various algorithms on Graphics Processing Unit (GPU) and a Central Processing Unit (CPU) coded in Java. We found that for the algorithms used, the CPU performed better as compared to the GPU. The intuition behind the results achieved is that since the algorithms run were not graphics-based and we didn't use parallelization, no improvement was achieved via the GPU.

2. Introduction:

In this section we describe the configurations/architecture of the 2 machines we used to compare performance of the various algorithms. In Section 2.1 we describe the PS3 and in Section 2.2 we briefly give the description of the configuration of the PC we used.

2.1 PlayStation 3 Architecture and Configuration

For this project we used a PS3 as the GPU. The architecture of the PS3 as it exists today was developed by: Toshiba, Sony and IBM. Cell is an architecture for high performance distributed computing. It is comprised of hardware and software Cells, software Cells consist of data and programs. These are sent out to the hardware Cells where they are computed, the results are then returned. GPUs claim to deliver similar or higher sustained performance in many non-graphical applications [1]. The difference between a Cell and a GPU is that a Cell is more general purpose and can be used on a variety of tasks. The motivation for this project came from this very fact. The PS3 was running Fedora Core 7 but we ran all of our tests via the terminal, not loading the GUI. This was done by modifying /etc/inittab.

The Cell architecture also has the added advantage of scalability. The 3.2 GHz (processor speed) Cell has the theoretical computing capability of 256 GFLOPS (Billion FP operations/sec). The Cell architecture was designed in a way that its slowest components wouldn't be hindered by Amdahl's Law [2].

The specifications of an individual hardware Cell are as follows (see Figure 1):
1) 1 Power Processor Element (PPE)
2) 8 Synergistic Processor Elements (SPEs)
3) Element Interconnect Bus (EIB)
4) Direct Memory Access Controller (DMAC)
5) 2 Rambus XDR memory controllers
6) Rambus FlexIO interface

The PS3 we used was part of the CECHAxx series. This series of models consists of a 60GB hard disk, 90nm CPU and GPU processes and 180W power consumption. It has a memory capacity of 256 MB of 700 MHz GDDR3 [3]. The PS3 has an installed RAM of 256MB. The memory bus bandwidth in keeping with the theoretical computing capacity is 25.6GB/sec. The RAM speed is 700 MHz. Graphics processing is handled by the NVIDIA RSX 'Reality Synthesizer' [4], which can output resolutions from 480i/576i SD up to 1080p HD.

2.1.1 Power Processor Element (PPE)-

The PPE is just like a CPU scheduling tasks for the multiple SPEs. The PPE runs the OS and applications are sent to the SPEs. The PPE is based on the Power Architecture used in the PowerPC processors. The PPE is a dual-issue, dual

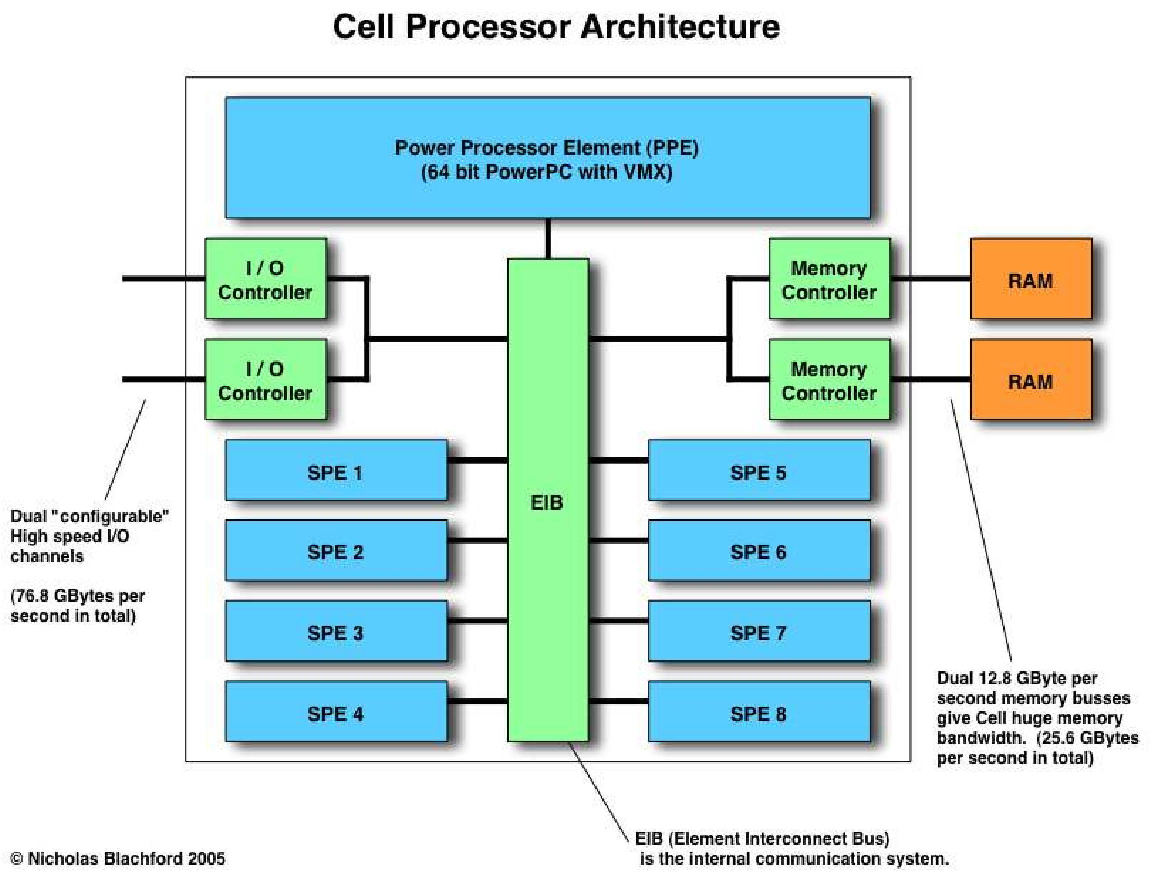
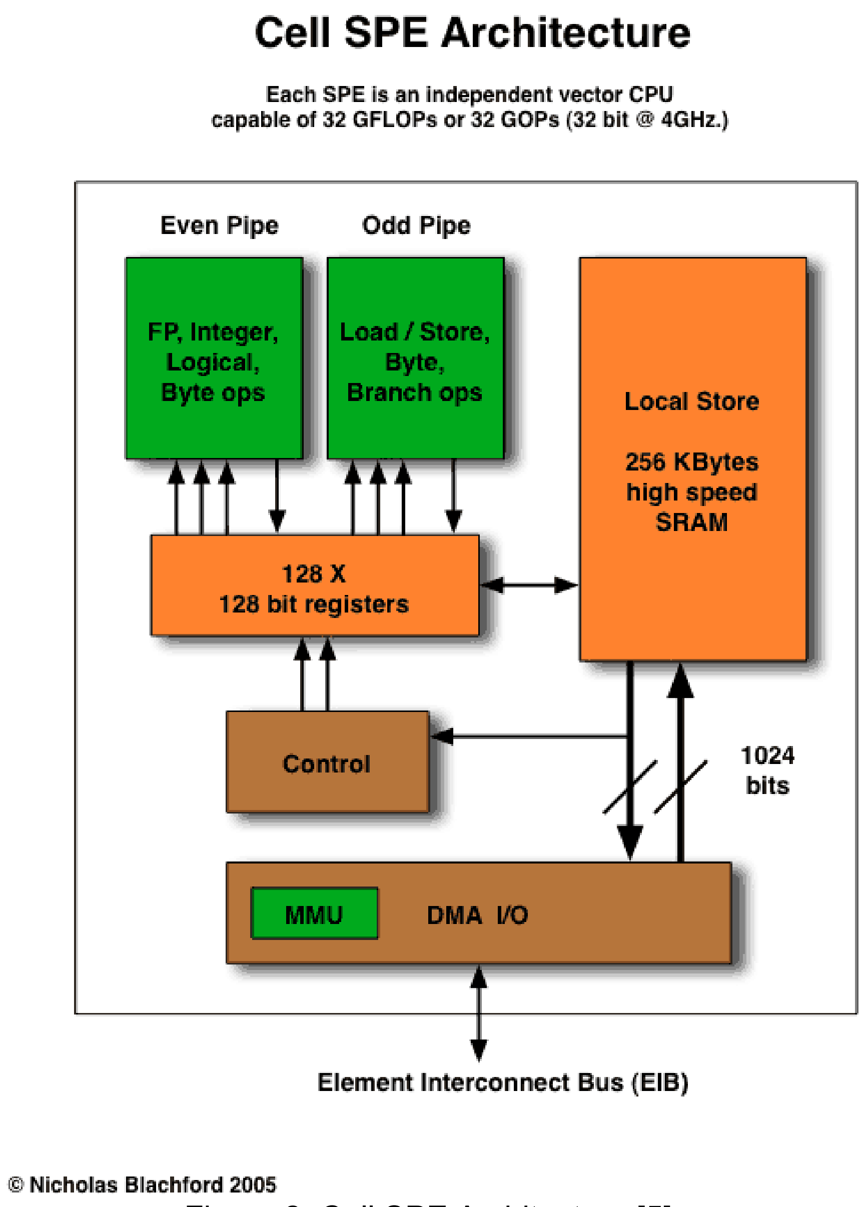

Cell Processor Architecture
Power Processor Element (PPE)
(64 bit PowerPC with VMX)
I / O Controller
I / O Controller
Memory Controller
Memory Controller
RAM
RAM
EIB
SPE 1
SPE 2
SPE 3
SPE 4
SPE 5
SPE 6
SPE 7
SPE 8
Dual "configurable" High speed I/O channels
(76.8 GBytes per second in total)
Dual 12.8 GByte per second memory busses give Cell huge memory bandwidth. (25.6 GBytes per second in total)
EIB (Element Interconnect Bus) is the internal communication system.
© Nicholas Blachford 2005

Figure 1: Cell Processor Architecture [5]

threaded, in-order processor and contains a simple instruction set (RISC system). The PS3 also has the advantage of being able to run multiple OS simultaneously using IBM’s hypervisor technology.

2.1.2 Synergistic Processor Elements (SPEs)-

Each Cell consists of 8 SPEs. The eighth SPE is disabled to improve chip yields. Only six of the seven SPEs are accessible to developers as the seventh SPE is reserved by the console's operating system. An SPE is a self contained vector processor (SIMD) which acts as an independent processor. As can be seen from Figure 2, each SPE contains 128 128-bit registers, 4 single-precision FP units working at 32 GFLOPS and 4 integer units working at 32 GOPS at 4GHz. The SPEs include a 256KB local store, not a cache. The local store negates the complexity of a caching mechanism and since there is no coherency mechanism, the design is further simplified. The local stores can move data at 64 GB/s for up to 10000 cycles without going to RAM. Both the PPE and the SPEs are in-order processors. The compiler is optimized to give the same performance as out-of-order functionality.


Cell SPE Architecture
Each SPE is an independent vector CPU
capable of 32 GFLOPs or 32 GOPs (32 bit @ 4GHz.)
Even Pipe
Odd Pipe
FP, Integer, Logical, Byte ops
Load / Store, Byte, Branch ops
Local Store
256 KBytes high speed SRAM
128 X 128 bit registers
Control
1024 bits
MMU
DMA I/O
Element Interconnect Bus (EIB)
© Nicholas Blachford 2005
Figure 2: Cell SPE Architecture [5]

Multiple SPE units within a Cell processor can be chained together to act as a stream processor [6]. The Cell processor can have multiple SPEs working on different parts of a task [7]. The first SPE reads in the input into its local store, performs the processing and stores the result back into its local store. The second SPE reads the output from the first SPE's local store and processes it and stores it into its local store and the process continues with the other SPEs.

2.1.3 EIB and DMAC-

The DMAC controls memory access but it doesn't control memory protection. 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. The original 1024-bit bus exists in between the I/O buffer and the local stores. The memory protection was replaced completely by MMUs and moved into the SPEs.

2.1.4 Programming model-

The Cell architecture has removed a level of abstraction which allows programmers to use all the 128 registers and 256KB local store. The Cell's physical hardware is equivalent to a Java "virtual machine" in the sense that it provides abstraction such that code written for a particular OS can be run on any other OS because the hardware executing it remains the same.

2.1.5 The GNU Compiler for Java

Although Sun provides Java compilers for a wide variety of OS including Windows, Solaris, Linux and Apple (OS X), it doesn't provide Java compilers for the PowerPC architecture. As a consequence of this, the GNU community built GCJ which is the GNU Compiler for Java. GCJ is a portable, optimizing, ahead-of-time compiler for the Java Programming Language. It can compile Java source code to Java bytecode (class files) or directly to native machine code, and Java bytecode to native machine code. Compiled applications are linked with the GCJ runtime, libgcj, which provides the core class libraries, a garbage collector, and a bytecode interpreter. libgcj can dynamically load and interpret class files, resulting in mixed compiled/interpreted applications. It supports most of the JDK 1.4 libraries plus some 1.5 additions. As is obvious from the previous statement, this compiler is still quite a basic compiler and doesn't include many of the new functionalities, syntax and operations available with J2SDK 1.5 and Java SE 1.6. This was one of the reasons the multi-threaded code we ran on the PC didn't run on the PS3.

2.2 PC Configuration

We used a multi-core (4 total) PC running Windows XP SP2 with 2.4GHz Intel Core2 Quad, 64 KB L1 Cache, 4096 KB L2 Cache and 267MHz bus speed. The machine was not hyperthreaded. The NVIDIA nForce 500 Series was responsible for the graphics processing [8]. For our tests, we ran the code via the command prompt. It was hoped that this would increase performance as compared to running the code via an Interactive Development Environment (IDE) like Eclipse, NetBeans, JCreator etc.

3. Background:

The following algorithms were used to compare the GPU and the CPU:

i) Matrix multiplication (3 types)
ii) Matrix inverse or pseudoinverse
iii) LU decomposition
iv) QR decomposition
v) Singular value decomposition
vi) Eigen value decomposition
vii) Sorting – quicksort, merge sort, radix sort

3.1 Matrix multiplication

Formally matrix multiplication for 2 matrices A and B such that
$A \in R^{m*n}$, $B \in R^{n*p}$
is defined as

$$(AB)_{ij} = \sum_{r=1}^{n} A_{i,r} B_{r,j} \qquad (1)$$

$\forall$ pair i and j where $1 \leq i \leq m$ and $1 \leq j \leq p$

The computational complexity of matrix multiplication between 2 matrices each of size n*n is $O(n^3)$, and 2 matrices of sizes m*n and n*p is O(mnp). This complexity can be improved by using certain optimizations to the loop ordering. We investigate 2 such loop orderings and compare performances.

3.1.1 Naïve Matrix Multiplication

The algorithm as stated in equation (1) above is the naïve matrix multiplication algorithm. It can be considered as the (i,j,k) loop ordering as the loops are all taken in order. The inner loop is an inner product of a row and a column. As stated above this straightforward approach has the computational complexity $O(n^3)$.

Loop construct:

```
for(int i = 0; i<m;i++){
        for(int j = 0;j<p;j++){
                for(int k = 0;k<n;k++){
                        C[i][j] += A[i][k]*B[k][j];
                }
        }
}
```

3.1.2 (j,i,k) JAMA loop ordering

The first loop ordering optimization is to interchange the first and second loops. JAMA [9] is a basic linear algebra package for Java. It provides user-level classes for constructing and manipulating real dense matrices. It is meant to provide sufficient functionality for routine problems. It is intended to serve as the standard matrix class for Java. JAMA is composed of six Java classes where JAMA's matrix multiplication algorithm, the A.times(B) algorithm, is part of the Matrix class. In this algorithm the resulting matrix is constructed column-by-column, loop-order (j,i,k). We also run a multi-threaded version of this loop optimization code using new functionality provided by Java SE 1.6.

Loop construct:

```
for(int j = 0; j < p; j++){
        for(int k = 0; k < n; k++){
                Bcolj[k] = B[k][j];
        }
        for(int i = 0;i<m;i++){
                double[] Arowi = A[i];
                double s = 0;
                for(int k = 0;k<n;k++){
                        s += Arowi[k]*Bcolj[k];
                }
                C[i][j] = s;
        }
}
```

3.1.3 (i,k,j) Pure row-oriented loop order

In this implementation the second and third for-loops are interchanged compared with the naïve implementation. These algorithms are the more efficient algorithms. The pure row-oriented algorithm does not traverse the columns of any matrices involved and we have eliminated all unnecessary declarations and initialization.

Loop construct:

```
for(int i = 0; i < m; i++){
        Arowi = A[i];
        Crowi = C[i];
        for(int k = 0; k < n; k++){
                Browk = B[k];
                double a = Arowi[k];
                for(int j = 0; j < p; j++){
                        Crowi[j] += a*Browk[j];
                }
        }
}
```

3.2 Matrix Inverse

The inverse of a square matrix A is a matrix A^{-1} such that
$A^* A^{-1} = I$ where I is the identity matrix
A square matrix is invertible if and only if the determinant of $A \neq 0$.
For a non-square matrix, we define a pseudoinverse. A pseudoinverse of a matrix A is a matrix that has some of the properties of the inverse matrix of A but not necessarily all of them. The term pseudoinverse commonly refers to the Moore-Penrose pseudoinverse [10-11].

The pseudoinverse A^+ of an m-by-n matrix A (whose entries can be real or complex numbers) is defined as the unique n-by-m matrix satisfying all of the following four criteria:

1) $A.A^+.A = A$ (AA^+ need not be the general identity matrix, but it maps all column vectors of A to themselves);
2) $A^+.A.A^+ = A^+$ (A^+ is a weak inverse for the multiplicative semi group);
3) $(A.A^+)^* = A.A^+$ (AA+ is Hermitian)
4) $(A^+.A)^* = A^+.A$ (A+A is also Hermitian).

Here M* is the Hermitian transpose (also called conjugate transpose) of a matrix M. For matrices whose elements are real numbers instead of complex numbers, M* = MT.

For the purpose of this project, we compute the inverse of square matrices and the pseudoinverse of non-square matrices. This is done via a call to the inverse function, part of the JAMA Matrix library.

3.3 Matrix LU Decomposition

The LU decomposition is a matrix decomposition which creates a matrix as the product of a lower triangular matrix and an upper triangular matrix.
A = LU

$$\begin{bmatrix} a1 & a2 & a3 \\ a4 & a5 & a6 \\ a7 & a8 & a9 \end{bmatrix} = \begin{bmatrix} l1 & 0 & 0 \\ l4 & l5 & 0 \\ l7 & l8 & l9 \end{bmatrix} \begin{bmatrix} u1 & u2 & u3 \\ 0 & u5 & u6 \\ 0 & 0 & u9 \end{bmatrix}$$

L will have zeros above the diagonal while U while have zeros below the diagonal. This decomposition is used in numerical analysis to solve systems of linear equations or calculate the determinant. A general n by n matrix can be inverted using LU decomposition. For the purpose of this project, we compute the LU decomposition by calling the LU function is the JAMA Matrix library.

3.4 Matrix QR Decomposition

A QR decomposition of a matrix is a decomposition of a matrix into an orthogonal and a right triangular matrix. For a square matrix a QR decompostion
A = QR
where Q is an orthogonal matrix and R is an upper triangular matrix.
For a Rectangular matrix

$$A = QR = Q \begin{bmatrix} R1 \\ 0 \end{bmatrix} = [Q1\ Q2] \begin{bmatrix} R1 \\ 0 \end{bmatrix} = Q1.R1$$

This matrix decomposition can be used to solve systems of linear equations. We compute the QR decomposition by calling the qr function which is part of the JAMA Matrix library.

3.5 Singular value decomposition

Singular value decomposition is a factorization of a rectangular real or complex matrix. The theorem states that "M is an m-by-n matrix whose entries come from the field K, which is either the field of real numbers or the field of complex numbers. Then there exists a factorization of the form

$$M = U\Sigma V^* \text{ ''}$$

U is an m-by-m unitary matrix over K
Σ is m-by-n diagonal matrix with nonnegative real numbers on the diagonal
V^* denotes the conjugate transpose of V, an n-by-n unitary matrix over K.
We perform this via a call to the svd function which is part of the JAMA Matrix library.

3.6 Eigen value decomposition

Eigen value decomposition is the factorization of a matrix into a canonical form. Here the matrix is represented in terms of its eigenvalues and eigenvectors. We perform the eigenvalue decomposition by calling the eig function part of the JAMA Matrix library.

3.7 Sorting Algorithms

We used the following sorting algorithms
(i) Merge Sort [13]
(ii) Quicksort [14]
(iii) Radix Sort [15]

3.7.1 Merge Sort

Merge sort is a sorting algorithm with a time complexity of O(n log n). The algorithm divides the unsorted list into two sublists of about half the size. It then sorts each sublist recursively by reapplying merge sort. Thus the two sublists that were created are essentially divided into two further sublists. It finally merges the two sublists back into one sorted list.

The principle behind this algorithm is that it will take fewer steps to sort a smaller list than a large list. Also fewer steps are required to construct a sorted list from two sorted lists rather than two unsorted lists. If a list is already sorted then traversing the list once takes fewer steps than if the list wasn't already sorted.

Pseudo code:

```
function merge(data, first, n1, n2)
copied = 0                    // number of elements copied from data to temp
copied1 =0                    //number of elements copied from first half of data
copied2 = 0                   // number of elements copied from second half of data
while (copied <n1) and (copied2 < n2)

      if  data[first + copied1] < data[first + n1 + copied2]
        temp[copied++] = data[first + (copied1++)]
      else
```

```
    temp[copied++] = data[first + n1 + (copied2++)]
```

The remaining entries in the left and right sub arrays are copied

```
while copied1 < n1
    temp[copied++] = data[first + (copied1++)];

while copied2 < n2
    temp[copied++] = data[first + n1 + (copied2++)];
```

Merge sort has an average and worst case performance of O(n log n).

3.7.2 Quicksort

Quick sort is a sorting algorithm with O(n log n) time complexity. It makes these many comparisons to sort N items. However in the worst case it takes O(n^2) comparisons. Typically quick sort is faster than other O(n log n) algorithms as its inner looop can be efficiently implemented on most architectures and in most real world data. It is a comparison sort.
Quick sort picks an element called a pivot from the list. It reorders the list so that all the elements, which are less than the pivot, come before the pivot and all the elements that are greater than the pivot come after it. After this partitioning the pivot is in its final position and this is called the partition operation. The sublist of lesser elements and the sublist of greater elements are sorted recursively.

Pseudo Code:

```
function quicksort (a, left, right)       // Sorts the array from a[left] to a[right]

i = partition(a, left, right)                    // i  is the partition computed
quicksort (a, left, i-1)
quicksort (a, i+1, right)

function partition(a, left, right)
i = left-1
j = right

while (true)
{
        while (lesserof(a[++i], a[right]))             // find item on left to swap
                                                              // a[right] acts as sentinel
                while (lesserof(a[right], a[--j]))      // find item on right to swap
                        if (j == left)  break;                 // don't go out-of-bounds

                if (i >= j) break;                         // check if pointers cross
                  swap(a, i, j);                       // swap two elements into place
 }
  swap(a, i, right)                                   // swap with partition element
  return i;
}
```

3.7.3 Radix Sort

Radix sort is a sorting algorithm that sorts integers by processing individual digits by comparing individual digits sharing the same significant position. The integers that are processed by this sorting algorithm are often called keys. The computational complexity of radix sort is O(nk/s) where k is the size of each key and s is the chunk size used by the implementation.

Pseudo code:

```
function Radix_Sort (int [ ] arr) {

    int [ ][ ] np = new int[arr.length][2];      //Empty array created to hold the split
                                                             by  digit
    int [ ] q = new int[0x100];
    int i, j, k, l, f = 0;

    for ( k=0; k<4; k++) {

        for ( i=0; i<(np.length-1) ;i++)
            np[i][1] = i+1;
        np[i][1] = -1;

        for( i=0; i<q.length; i++)
            q[i] = -1;

        for( f=i=0; i<arr.length; i++) {
            j = ((0xFF<<(k<<3)) & arr[i])>>(k<<3);
            if ( q[j] == -1)
                l = q[j] = f;
            else{
                l = q[j];
                  while( np[l][1] != -1)
                        l = np[l][1];
                  np[l][1] = f;
                 l = np[l][1];
            }
            f = np[f][1];
            np[l][0] = arr[i];
            np[l][1] = -1;
        }

        for( l=q[ i=j=0 ]; i<100 ; i++)

                    for( l=q[i]; l != -1; l = np [l ][1])
                      arr[ j++ ] = np[ l ][ 0 ] ;
    }
  }
```

4. Results:

In this section we describe the results obtained after running the algorithms described in Section 3. First we look at the performance of the matrix operations and then the sorting algorithms.

Figures 3-4 display the performance statistics for the 3 types of matrix multiplications; namely the naïve matrix multiplication (section 3.1.1), the JAMA loop ordering (section 3.1.2) and the pure row-oriented loop ordering (section 3.1.3). For the PC we also ran the multi-threaded version of the JAMA loop ordering. We couldn't run this code on the PS3 because of the absence of Java SE 1.6 libraries on the GCJ compiler. As can be seen from the plots, the performance for each implementation is respectively faster on the PC as compared to the PS3. In terms of the best performing algorithm, the multi-threaded version of the JAMA loop ordering has the least execution time. The second-best algorithm (and best in case of the PS3) is the pure row-oriented loop ordering. The reason for this is that in case of the pure row-oriented loop ordering we are accessing the same object array several times (when traversing a row) whereas for the other implementations we are accessing different object arrays when traversing columns.

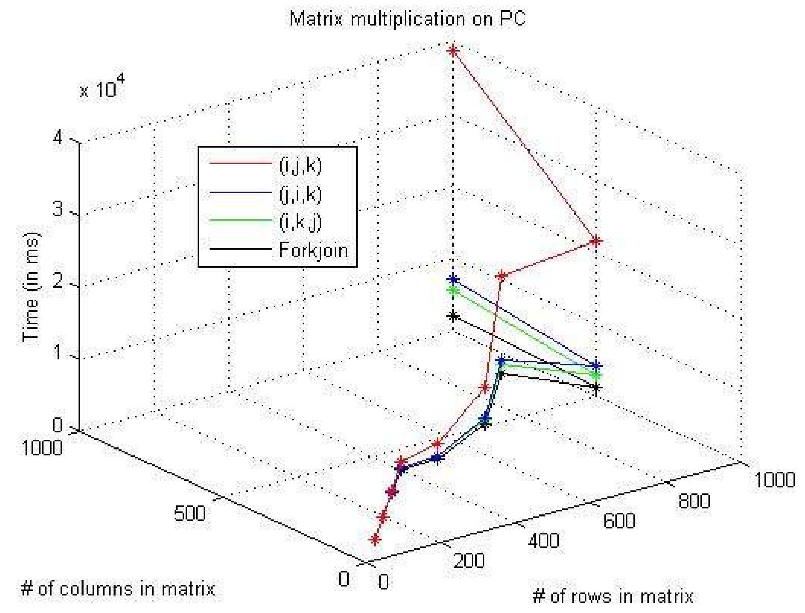


Figure 3: Matrix multiplication statistics on PC

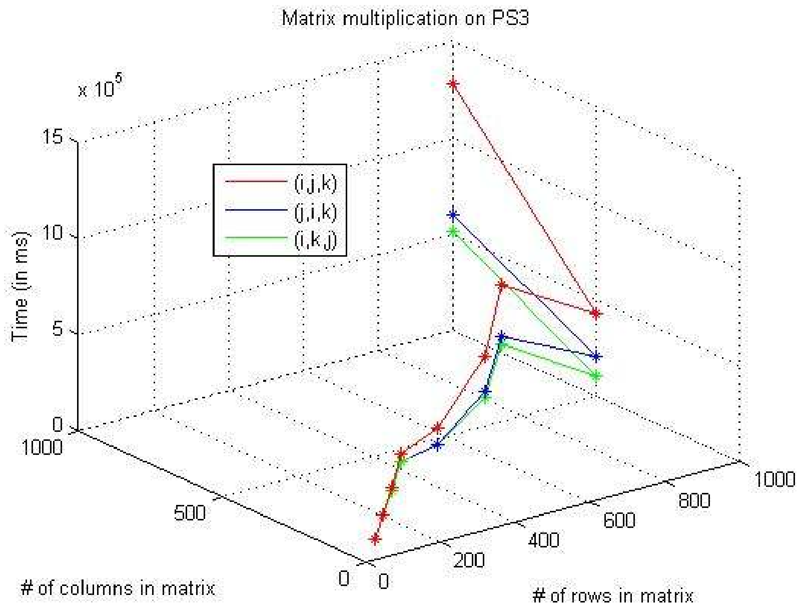


Figure 4: Matrix multiplication statistics on PS3

An interesting observation was made was while multiplying a vector and a matrix. Given a vector A and a matrix B, the execution times for A'*B and B*A are not the same. The number of operations (multiplications and additions) for both is the same yet there is a difference in the execution times. We intuit that this change is a result of the cache access. This result would be true if the elements in cache were accessed row-wise. Figures 5-6 display this result for PC and PS3. The performance of the multiplication is faster on the PC as compared to the PS3.

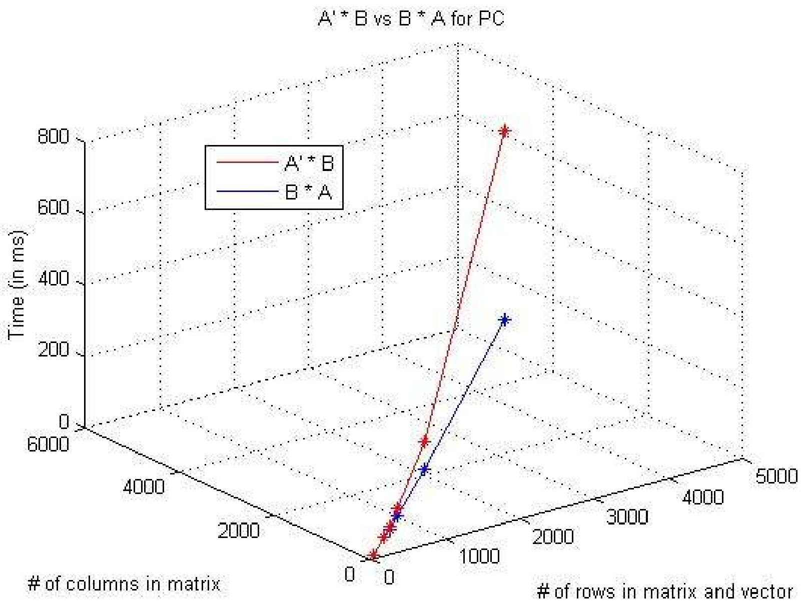


Figure 5: Vector and matrix product on PC

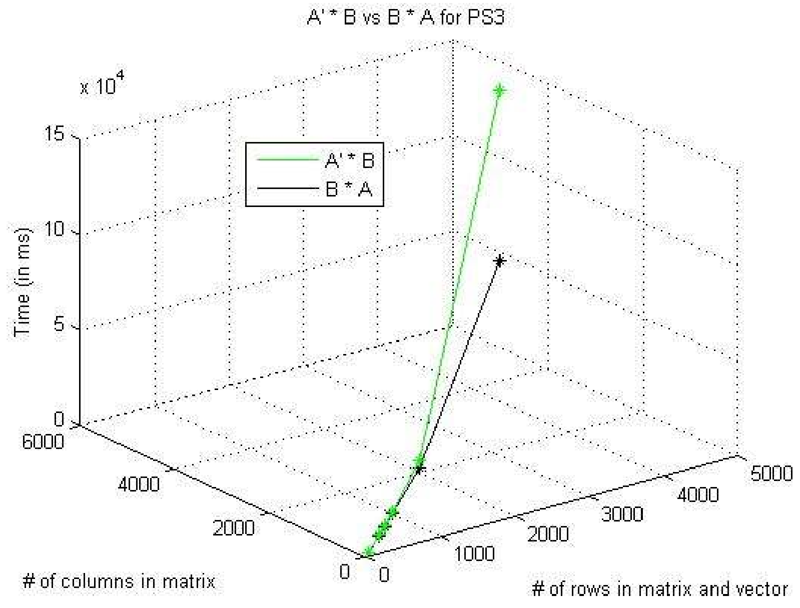


Figure 6: Vector and matrix product on PS3

For the matrix inverse, we computed the inverse of the square matrix or the pseudoinverse of the non-square matrix. We also calculated the time it takes to compute the LU and QR decompositions of the matrix. Figures 7-8 display the statistics for the matrix inverse, LU and QR decompositions.

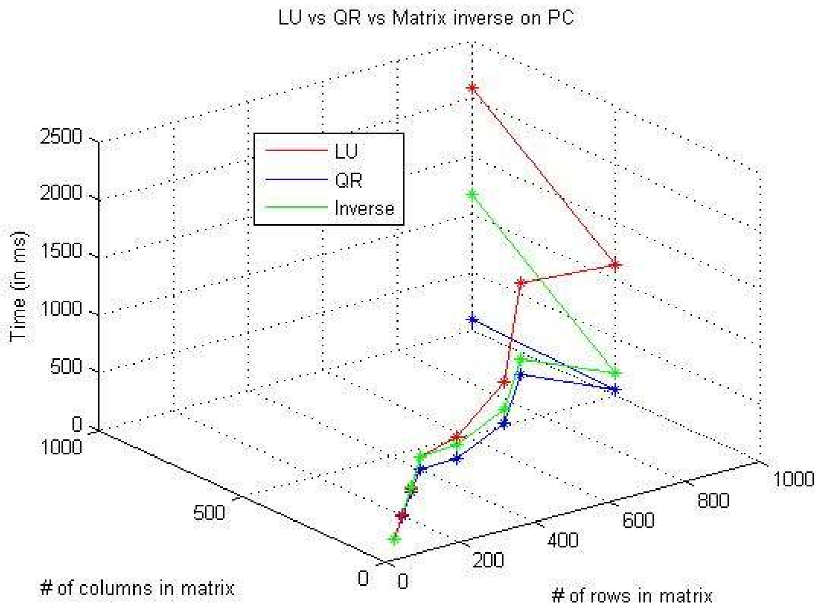


Figure 7: Matrix inverse, LU and QR decompositions on PC

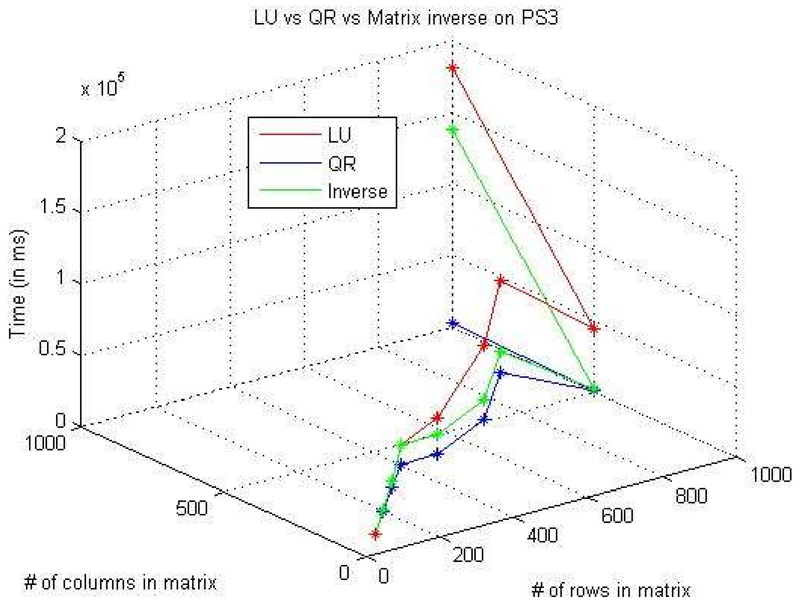


Figure 8: Matrix inverse, LU and QR decompositions on PS3

As can be seen from the figures, the PS3 is much slower as compared to the PC in computing the 3 values. Of the 3 quantities, the QR decomposition takes significantly lesser execution time than LU decomposition and the matrix inverse. This could be because the QR decomposition is only computing the determinant of the matrix whereas the LU decomposition and the matrix inverse actually compute the inverse of the matrix.

We also computed the Singular Value and Eigen Value Decompositions. As can be seen from Figures 9-10, the PC performs better at computing these 2 quantities than the PS3.

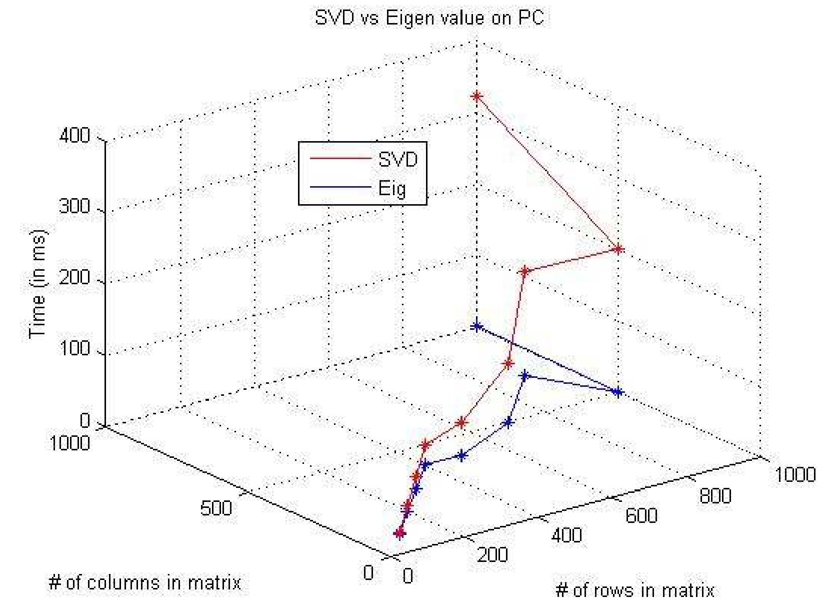


Figure 9: Singular Value and Eigen Vale Decompositions on PC

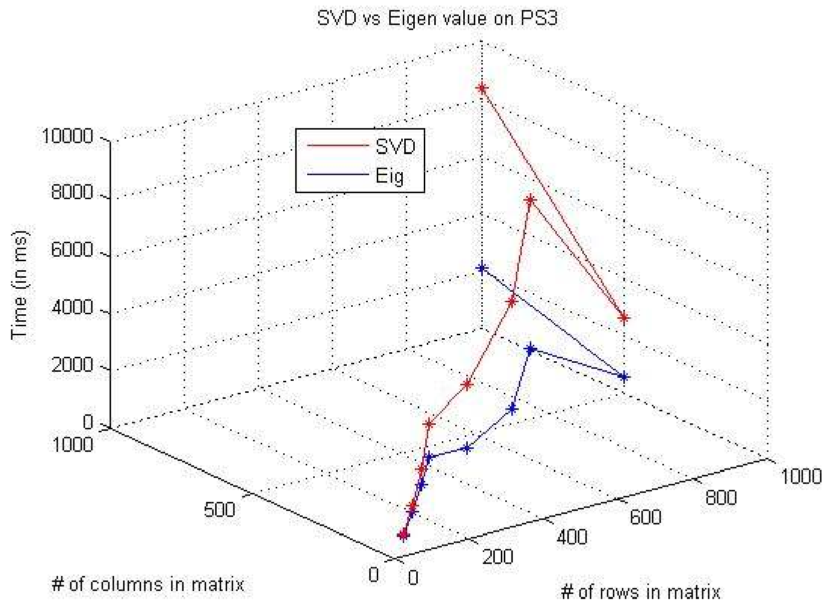


Figure 10: Singular Value and Eigen Value Decompositions on PS3

Figures 11-13 display the performance statistics for the PC and the PS3 for merge sort, quick sort and radix sort respectively. For smaller-sized inputs (<=10000), we ran the algorithm for 1000 trials and averaged the results. For larger-sized inputs (>10000) we ran the algorithms for 100 trials and averaged the result. As can be seen from the graphs, the PC performs better than the PS3 in terms of the input array size possible (memory) and the run time for the algorithms.

This is an interesting observation that the PS3 was slower as compared to the PC on these tasks. A reasonable explanation for this phenomenon is what we observed in the architecture of the PS3. There are 6 SPEs, each with a 256KB local store. Taken together the total capacity of the local stores comes out to 1536KB. This is still less as compared to the 4096KB L2 cache for the PC. Thus for larger sized inputs, thrashing was a common phenomenon for the PS3 and the following warning was seen frequently:
"GC Warning: Repeated allocation of very large block (appr. size 40001536): May lead to memory leak and poor performance"
Another point to note is that the tasks must be parallelized to make use of all 6 SPEs and only then do we see a performance improvement. Since we didn't parallelize our code, we were unable to get the advertised performance boosts.

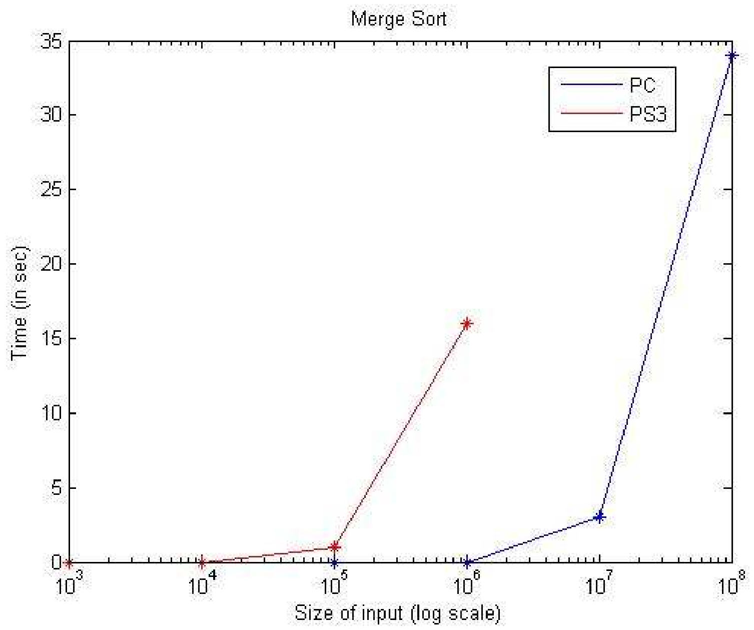


Figure 11: Merge sort performance statistics for PC vs. PS3

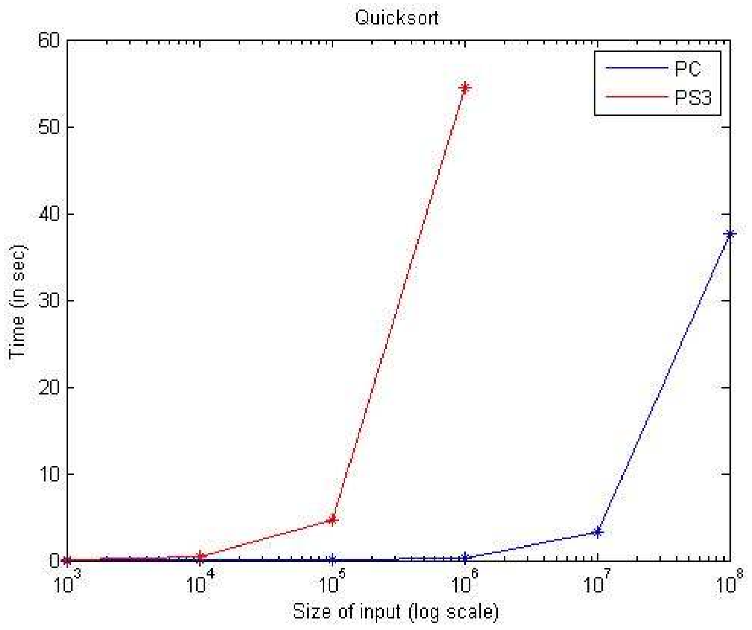


Figure 12: Quick sort performance statistics for PC vs. PS3

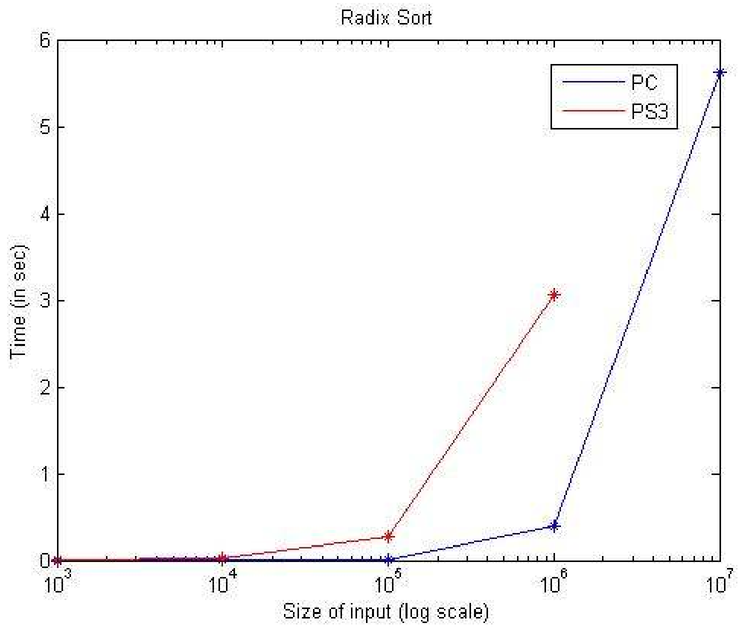


Figure 13: Radix sort performance statistics for PC vs. PS3

A general observation about the algorithms was that radix sort performed better than quick sort and merge sort at smaller-sized inputs but was unable to complete execution for larger-sized inputs. The reason we intuit for this is the basis for the algorithm which compares the inputs digit-by-digit. As the size of the input number grows, this comparison grows exponentially and hence memory heap size issues occur.

The individual values for all algorithms on both the PC and the PS3 are given in Table 1 in the Appendix.

5. Conclusions:

This project looked at the performance statistics of a PC vs. the PS3 and found that without parallelization, the PS3 offers no benefit over the PC. This also means that the RISC architecture did not provide a performance benefit over the CISC architecture. Multiple algorithms were run on both machines to verify this result including matrix operations like multiplication, inverse, decompositions etc and sorting algorithms like merge sort, quicksort and radix sort. It was also found that the Java compiler for PowerPC architecture (of which the PS3 is a type) is seriously lacking in functionality and optimization as compared to Sun J2SDK 1.5 or Java SE 1.6.

6. References:

[1] GPUs have been measured at 10 times faster than conventional CPUs http://www.gpgpu.org/vis2004/A.lefohn.intro.pdf

[2] G. Amdahl. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. AFIPS Conference Proceedings (30): 483-485. (1967)
[3] GDDR3 specification http://www.jedec.org/download/search/3_11_05_07R15.pdf
[4] Reality Synthesizer (RSX) http://en.wikipedia.org/wiki/RSX_%27Reality_Synthesizer%27
[5] Images taken from http://www.blachford.info/computer/Cell/Cell1_v2.html
[6] IBM's Hypervisor used to validate the Cell http://www.research.ibm.com/hypervisor
[7] W.J. Dally, U.J. Kapasi, B. Khailany, J.H. Ahn, A. Das Stream Processors: Programmability and Efficiency
[8] nForce 500 Series for Intel http://www.nvidia.com/page/nforce5_intel.html
[9] J. Hicklin, C. Moler, P. Webb, R. F. Boisvert, B. Miller, R. Pozo, K. Remington. JAMA: A Java matrix package. http://math.nist.gov/javanumerics/jama [5 May 2003].
[10] E. H. Moore. On the reciprocal of the general algebraic matrix. Bulletin of the American Mathematical Society 26: 394–395. (1920)
[11] R. Penrose. A generalized inverse for matrices. Proceedings of the Cambridge Philosophical Society 51: 406–413. (1955)
[12] G. Gundersen, T. Steihaug. Data Structures in Java for matrix computations. Concurrency Computat.: Pract. Exper. 2004; 16:799–815. (2004)
[13] D.E. Knuth. Sorting by Merging. §5.2.4 in *The Art of Computer Programming, Vol. 3: Sorting and Searching, 2nd ed.* Reading, MA: Addison-Wesley, pp. 158-168, 1998.
[14] R. Sedgewick. Implementing Quicksort Programs. *Comm. ACM* **21**, 847-857, 1978.

[15] A. Gibbons and W. Rytter. Efficient Parallel Algorithms. Cambridge University Press, 1988.

7. Appendix

Table 1 gives the values for matrix multiplication for the PC.

Rows	Columns	(i,j,k) (in ms)	(j,i,k) (in ms)	(i,k,j) (in ms)	Forkjoin (in ms)
100	100	8.64	3.26	2.65	2.34
200	200	80.34	25.91	24.67	9.07
300	300	352.09	99.55	97.73	32.03
400	400	1088.79	273	270.71	81.9
500	400	2293.23	546.59	488.47	155.05
700	500	5414.19	1206.44	963.79	334.71
900	700	14508.25	2849.74	2190.49	801.89
1000	500	21465.5	4236.26	2867.15	1179.1
1000	1000	38812.76	7156.35	5675.97	2035.38

Table 2 gives the values for matrix multiplication for the PS3.

Rows	Columns	(i,j,k) (in ms)	(j,i,k) (in ms)	(i,k,j) (in ms)
100	100	580.12	391.63	408.34
200	200	6264.81	3545.07	3704.23
300	300	28671.39	14217	14860.43
400	400	80776.13	39552.71	41318.31
500	400	162907	79042.26	74374.39
700	500	364849.74	178057.83	147199.92
900	700	490789.21	222415.12	181660.01
1000	500	427556.52	201790.61	103294.37
1000	1000	1285449.27	595615.33	515671.17

Table 3 gives the values for vector-matrix multiplication for the PC and the PS3.

Rows	Columns	A'*B (in ms) PC	B*A (in ms) PC	A'*B (in ms) PS3	B*A (in ms) PS3
100	100	0.16	0	7	4.33
500	500	3.28	0.62	239.93	110.89
700	700	9.38	1.71	725.39	323.79
1000	1000	24.74	4.19	1748.12	762.83
2000	2000	92.21	15.91	6085.16	2454.62
5000	5000	614.38	85.34	134674.21	44918.23

Table 4 gives values for LU, QR and matrix inverse for PC.

Rows	Columns	LU (in ms)	QR (in ms)	inverse (in ms)
100	100	1.26	0.16	1.4
200	200	12.98	0.62	10.76
300	300	40.92	3.89	41.71
400	400	104.2	7.31	111.54
500	400	193.57	13.43	125.81
700	500	382.83	24.17	144.28
900	700	840.53	41.48	171.52
1000	500	1136.36	53.88	195.48
1000	1000	2095.11	79.79	1169.91

Table 5 gives values for LU, QR and matrix inverse for PS3.

Rows	Columns	LU (in ms)	QR (in ms)	inverse (in ms)
100	100	153.21	18.43	152.23
200	200	1274.21	79.15	1287.32
300	300	5057.23	228.45	5106.36
400	400	13888.82	489.9	13993.87
500	400	26002.73	872.93	14441.75
700	500	54024.47	1598.14	15279.81
900	700	66682.64	2078.17	17045.14
1000	500	43689.68	1154.82	1706.03
1000	1000	181832.91	2825	138421.08

Table 6 gives SVD and Eigen value decompositions for the PC and the PS3.

Rows	Columns	SVD (PC)	SVD (PS3)	Eig (PC)	Eig (PS3)
100	100	1.53	50.13	0	23.17
200	200	8.1	245.42	0	71.69
300	300	16.4	698.73	0.16	176.64
400	400	29.06	1518.96	0.19	330.41
500	400	46.86	2512.06	0.22	412.22
700	500	82.1	4271.72	0.27	520.25
900	700	145.78	6202.54	0.31	1016.83
1000	500	200.2	2622.87	0.16	550.71
1000	1000	322.41	8362.09	0.47	2102.44

Table 7 gives the values for the sorting algorithms. All times are in seconds.

Input	Merge Sort (PC)	Merge Sort (PS3)	Quicksort (PC)	Quicksort (PS3)	Radix sort(PC)	Radix sort (PS3)
1000	0	0	0	0.028	0	0.002
10^4	0	0	0.002	0.387	0.001	0.023
10^5	0	1	0.024	4.622	0.013	0.268
10^6	0	16	0.255	54.508	0.387	3.063
10^7	3	NA	3.294	NA	5.626	NA
10^8	33.98	NA	37.618	NA	NA	NA