Proceedings of the IASTED International Conference
CIRCUITS, SIGNALS, AND SYSTEMS
November 28 - December 1, 2004, Clearwater Beach, FL, USA

PERFORMANCE OF HYPERSPECTRAL IMAGING ALGORITHMS USING ITANIUM ARCHITECTURE

Wilfredo Lugo-Beauchamp[1], Kennie Cruz[2], Carmen L. Carvajal-Jiménez[2], and Wilson Rivera[2]

[1] Software Solutions Group
Hewlett Packard Technology Center
Aguadilla, Puerto Rico, USA
Wilfredo.Lugo@hp.com

[2]Electrical and Computer Engineering Department
University of Puerto Rico, Mayagüez Campus
P.O.Box 9042, Mayaguez, Puerto Rico, USA
{Kennie.Cruz, Carmen.Carjaval,
Wilson.Rivera}@ece.uprm.edu

Abstract

This paper describes the experiences and results on implementing a set of hyperspectral imaging analysis algorithms on the Itanium Processor Family. On Itanium architecture all instructions are transformed into bundles of instructions and these bundles are processed in a parallel fashion by the different functional units. Experimental results show that exploiting implicit parallelism and linking HP Mathematical LIBrary optimized for Itanium yield significant improvement in performance.

Keywords: Itanium, IA64, Remote Sensing, Hyperspectral Imaging, Image Classifiers, HP-MLIB

1. Introduction

Sensors based on imaging spectrometry or so called hyperspectral imagers collect high spectral resolution data over a couple of hundred of wavelengths effectively producing an image where at each pixel we get the spectral response of the object(s) in the field of view of the sensor. Hyperspectral Imaging (HSI) analysis is based on the concept of imaging spectrometry where spectral and spatial information is used to identify or detect objects, or estimate parameters of interest. As the object of interest is embedded in a complex media (i.e. coastal waters or skin), the measured signature is a distorted version of the original object signature (e.g. a coral reef or a blood vessel) mixed with clutter. By large hyperspectral imaging analysis concentrates on dimensionality reduction and classification algorithms. Dimensionality reduction algorithms reduce the data volume (dimensionality), without loss of critical information, so that it can be processed efficiently. Classification of a hyperspectral image sequence, in turn, identifies which pixels contain various spectrally distinct materials.

Different classification metrics have been proposed from minimum distance, such as Euclidean, Fisher Linear Discriminant, and Malahanobis, to maximum likelihood [1] to correlation matched filter-based approaches such as spectral signature matching [2]. There are two major techniques to image classification: supervised and unsupervised. In supervised classification techniques, an analyst develops quantitative descriptions of the spectral characteristics of the various classes of interest for a particular scene. These descriptions are then used as reference spectral signatures against which every pixel in an image is compared. The pixels are classified according to the spectral signature they most closely resemble. In unsupervised classification, the algorithms do not use training data as the basis for classification. Instead, the algorithms used examine the unknown pixels in the image and aggregate them into various classes according to the clusters found in the spectral space that contains the image.

We have implemented a set of hyperpectral imaging analysis algorithms. These algorithms have been tuned and ported to be run on the Itanium Processor Family. We describe in this paper the experiences and results on implementing these algorithms. The structure of this paper is as follows. Section 2 provides an overview of hyperspectral imaging algorithms. Section 3 discusses implementation issues. Section 4 presents the results and discusses related work. Finally, section 5 draws conclusions and potential future work.

2. Background

A complete description of hyperspectral algorithms can be found elsewhere in [3]. In the next subsections we describe briefly each of the algorithms we implemented on Itanium.

2.1 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a procedure for transforming a set of correlated variables into a new set of uncorrelated variables. This transformation is a ariation. The second axe contains the maximum amount of variation orthogonal to the first axe. The third axe contains the maximum amount of variation orthogonal to the first and second axes and so on The data or hyperspectral image is represented by a matrix where each row represents a pixel or observation and e orthogonal basis for the covariance matrix can be obtained by finding its eigenvalues and eigenvectors. A total of N eigenvalues $\lambda_1, \lambda_2, \lambda_3, \ldots\ldots \lambda_{N-1}, \lambda_N$ are obtained from the covariance matrix. Each of these eigenvalues has a correspondent eigenvector that is orthogonal to the other vectors. If the eigenvalues are ordered from maximum to minimum the vector related to the biggest value contains the direction of the largest variance of the data. In this way we can find directions in which the data set has the most significant amount of energy.

Since the first K direction vectors represent a significant amount of energy of the whole set, we can reduce the dimensionality of the original observations or pixels by projecting each one of them to the first K orthogonal vectors. This reduces the dimensionality from N to K reducing considerably the complexity and computational workload.

2.2 Euclidean Distance Classifier (EUC)

In the classification area we want to obtain a thematic map that classifies each pixel of the image as a member of one specific class among C classes. The variable C is a parameter that establishes beforehand the number of classes in which each vector pixel can be classified. How this parameter is obtained depends totally on the region of interest and on the prior knowledge of the area. A classification algorithm takes C initial points on the image. In this case those points are considered the signatures of the materials we want to detect. The distances between each vector pixel on the image and the C vectors are calculated using the Euclidean metrics in equation (1), where **X** is the vector pixel, $\mathbf{M_i}$ is the mean vector for the class i and **N** is the number of bands.

$$\mathbf{g_i(X)} = (X - M_i)^T (X - M_i). \quad \mathbf{(1)}$$

Each vector pixel is then assigned as a member of the closest C point. Next, a new centroid is calculated

rotation of the original axes to new orientations that are orthogonal to each other and therefore there is no correlation between variables. In this new representation, the first axe accounts for the maximum amount of v

each column represents a wavelength or spectral band. Given an n-by-n image with N bands the data matrix has n^2 rows and N columns. Using this data matrix we can calculate the covariance matrix for the whole data. The covariance matrix, of dimension N-by-N, provides the relation between spectral bands. Th for each of the resultant C classes. Then the pixels are reclassified with the new centroids. The process of classification continues will until none of the vector pixels change from one class to another. The algorithm stops when there is not signifiacant change in the vector of pixels.

2.3 Maximum Likelihood Classifier (ML)

This classifier is based on statistical information. Assuming that the vector pixel X is normally distributed with mean μ and variance Σ, where both μ and Σ are unknown, the likelihood function becomes:

$$\mathbf{h_i(x)} = -\frac{1}{2}\ln\left|\hat{\Sigma}_i\right| - \frac{1}{2}(X - \hat{\upsilon}_i)^T \hat{\Sigma}_i^{-1}(X - \hat{\upsilon}_i). \quad \mathbf{(2)}$$

The vector pixel X belongs to the class that has the functions with the largest $\mathbf{h_i(X)}$. When the above equation is maximized and solved we have:

$$\hat{\upsilon}_i = \frac{1}{n_i}\sum_{k=1}^{n_i} X_k, \quad \mathbf{(3)}$$

$$\hat{\Sigma}_i = \frac{1}{n_i - 1}\sum_{k=1}^{n_i}(X_k - \hat{\upsilon}_i)(X_k - \hat{\upsilon}_i)^T. \quad \mathbf{(4)}$$

These mean and covariance have to be recomputed for every class. The algorithm stops when there is not a significantly change between the μ and Σ previously calculated.

2.4 Feedback Iterative Method (FIM)

The goal of the feedback iterative algorithm is to select the subset of bands that best separates the centroids of a given number of classes. This is done by creating the whole possible combinations of m bands from the total of N, where m is the desire final number of bands and N is the total number of bands. The covariance matrix and the

mean for each class are calculated for each set. These values can be obtained by using the pixel class membership of a previous classification. This classification can be an initial one when the algorithm is starting or a classifier output on a previous iteration.

$$\binom{N}{m} = \frac{N!}{(N-m)!m!}. \qquad (5)$$

Among all the sets the one with the largest average distance between its class centroids is selected. The selected set is the input to the classifier and when it finishes computing the classified pixels are used again to select other possible sets. The algorithms stop when the same set is continuously selected or the algorithm reaches its maximum iteration number.

3. Implementation Issues

The algorithm implementations use two different libraries: ATLAS + CLAPACK for IA32 and HP MLIB for IA64.

Automatically Tuned Linear Algebra Software (ATLAS) [4] focuses on applying empirical techniques in order to provide portable performance. Currently, it provides C and Fortran77 interfaces to a portably efficient BLAS [5] implementation, as well as a few routines from LAPACK [6]. ATLAS was compiled on IA32 systems using gcc compiler version 3.3-3 on an Intel© Xeon™ 2.2GHz machine. The ATLAS version used was 3.7.3. Most of ATLAS routines only provide a subset of LAPACK routines, so for the proper use of these routines CLAPACK [7] should be installed. CLAPACK is the same Fortran LAPACK library build using a FORTRAN to C conversion utility called f2c. fThe entire Fortran 77 LAPACK library is run thought f2c to obtain C code, and then modified to improve readability.

The HP Mathematical LIBrary (HP MLIB) [8] is a HP based high performance numerical package optimized for the IA64 and PA-RISC architectures. This package consists of three packages LAPACK, VecLIB [9] and SCILIB [10]. VECLIB contains the complete set of BLAS routines.

Using these libraries we have a common set of BLAS routines that could be implemented on the current code to obtain benchmarks based on optimized libraries. The original code was modified to replace original code calls for BLAS routines. The same BLAS routine was used on both architectures. Pre-compiler flags (#define) were added on the code so we are able to differentiate the calls depending on the architecture and the type and number of parameters parameters. In order to avoid compiler errors and increase flexibility we have created two different make-files: Makefile.ia32 and Makefile.ia64. Depending on which architecture a symbolic link should be made.

Original Function	**Description**	**BLAS Function**
Jacobi()	Calculates all EigenVectors and EigenValues of a symmetric matrix	sytrd() – convert a martix into its tridiagonal form stevx() – calculate eigenvectors and eigenvalues of a tridiagonal matrix
matmat()	Performs a matrix-matrix multiplication	gemm()
vecmatmul()	Peforms a vector matrix multiplication	gemv()

Table 1. Function replace using BLAS routines

The codes were run on a Intel© Xeon™ 2.2GHz machine running Red Hat 8.0 with 1GB of RAM on the IA32 side and for IA64 we use a HP rx4640 machine with one IA64 Madison processor 1.5Ghz and 6MB of cache running Red Hat Advanced Server 2.1 with 1GB of RAM. On IA64 we used the Intel 8.0 non commercial compiler and on IA32 we used the gcc 3.3-3 compiler. For code profiling we used the gprof tool [11].

4. Results and Related Work

4.1 Benchmarks

One of our main goals in this research besides exploring the IA64 capabilities was to test the mathematical libraries available for the Itanium architecture. So to check these libraries we developed all the mathematical functions on our own and then ported the ones with the biggest performances issues. Not surprisingly the routines with several performance penalties were the matrix to matrix multiplication, eigenvector and eigenvalues calculation and vector to matrix multiplication. In Table 1 we summarize the functions and also their counterparts on the BLAS library. People familiar with the BLAS library know that there are a lot of routines available that could replace our original functions. These BLAS routines were selected basically because the matrix used on HSI are in general real matrices and most of the calculations are done using symmetric matrices.

In Table 2, we present the execution times obtained for different versions of the algorithms. The first two columns show the execution time of the algorithms using our own mathematical functions. The two columns of the left show the algorithm after the BLAS routines were used.

On the Principal Component Algorithm we can see an improvement of 50% without any major changes, just compiling the code. When we integrate BLAS routines we see a breakthrough of nearly 23 times faster on both architectures. On the Feedback Iterative Method we see an improvement of 26% on IA32 and a lousy 1% on Itanium. Also as the reader can notice execution on Itanium took almost an hour more to execute than on IA32 on both algorithm version. So the algorithm took a performance penalty of 37%. The story on the Euclidean distance classifier is completely different. We see an improvement of more than 2 times faster by just compiling the application on IA64. We also get a boost on performance of more than 4.5 times on IA32 by using BLAS routines and one of 11.7 times on IA64. In the Maximum Likelihood algorithm we encountered a penalty of 2.3 times in the IA64 executions. When we integrate BLAS to the code we then see an improvement of 43.8% on IA32 a 93.4% on IA64.

Algorithm	IA32	IA64	IA32 (Optimized)	IA64 (Optimized)
PCA	1m39s	1m9s	4.13s	3.18s
FIM	2h39m	3h38m	2h6m8s	3h36m58s
EUC	5m3s	2m17s	1m5.74s	11.66s
ML	29m8s	1h7m	16m22s	4m22s

Table 2. Algorithms benchmarks before and after BLAS library replacements.

4.2 Analysis

4.2.1 Principal Component

From the previous results there is not apparent performance benefit on the Itanium architecture. For this to happen we need to further analyze the algorithms. In Figure 1 we show the four major components of the PCA. The first block is gathering the spectral image from a file; since this task is mostly dependant on the disk I/O we do not cover it. The next block is the covariance calculation. Inside this block we get the image matrix and perform some manipulations. The most computing intensive task of this block is a matrix multiplication. This matrix multiplication accounted for more that 80% of the block. It explains the performance gains when we use BLAS routines. The last block is also a matrix multiplication. Consequently only the gemm() routine replacement basically eliminates the biggest hotspot on block two and almost eliminate the hotspot on block 4. On both BLAS implementations we can get similar performance benefits, so basically both libraries optimized the architectural calls. Using the same algorithm without any modifications we can see an improvement on Itanium. We believe this is mostly due the highest clock speed and some average usage of the Itanium cache.

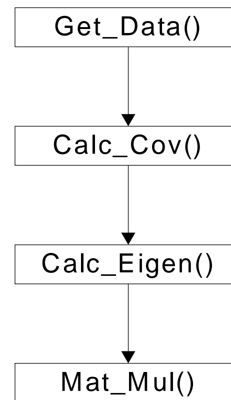

Figure 1. Principal Component Algorithm block components

4.2.2 Feedback Iterative Method

In the FIM algorithm the numbers of combinations generated increases exponentially depending on the number of spectral channels and the subset selected. In Table 3 we show a sample of the combinations generated depending on the different values using Equation (5).

Parameters		Combinations
N	S	
220	3	17,505,400
220	5	41,025,655,440
220	7	44,942,628,733,200

Table 3. Sample numbers of combinations generated for the specific parameters (N = number of spectral bands, s = numbers of elements in the combination)

A profiling on the FIM code shows that the means calculation is accounted for 76% of the time on Itanium and 86% on IA32. It is found that calc_means() function does not benefits from any of the BLAS routines implemented. The function goal is basically to get all pixels from a combination and based on its membership the mean or centroid for each of the classes is calculated. For every combination a new set of pixel values will be evaluated so there is very little opportunity for Itanium

cache usage. Although we know why the BLAS routines implementation on Itanium have very little impact we still are looking to found the cause in the differences in performance between IA32 and IA64 on both implementations. We noticed that calc_means() took an average of 1409 seconds per call to execute on Itanium and 323 seconds on IA32. This functions has several loops implemented with a $O(n^2)$ as its biggest point so the problem resides on the calc _means() procedure but further analysis is needed to determine what exactly is the compiler generated code for this function that leads to this performance problem on Itanium.

4.2.3 Euclidean Distance Classifier

The Euclidean distance classifier exhibits the best performance gains. It is because this algorithm exploits the Itanium cache benefit. On IA32 the Euclidean() function (the one in charge of the main distance calculation) is accounted for the 73% of the execution time with 47 seconds. On the IA64 the Euclidean routine is accounted for only 31% of the processing with a self call of 6.64 seconds. If we check Equation (1) we can see that the main calculation is an accumulative value (a multiplication and then an addition to the previous value). Itanium architecture is optimized for these types of operations. With its 3 levels of cache Itanium ranging from 3-9MB of cache in total, Itanium can outperform all other architectures in this type of sequential reads. Moreover, all the data for the whole algorithm can be available on cache for the whole executions. New distances are calculated using the same pixel vectors of the original image so there should be a small amount of cache misses requesting data. We assume most of the image data is stored on the nearest cache and then all the operations are executed, but the procedure is mostly the same for the whole algorithm.

4.2.4 Maximum Likelihood Classifier

On the Maximum Likelihood Classifier, although we have the same accumulative effect than on the Euclidean distance (Equation 2) it is now combined with a series of matrix manipulations. Moreover since the covariance is used, we need to calculate it for each class. On the original implementation the main function hotspot was the matrix multiplication on the covariance function. With the integration of the gemm() BLAS routine the performance benefits are tremendous. On the other hand we have a similar issue that with the FIM algorithm. When the code was compiled on Itanium without any modification, the performance penalty was prohibited. The matrix multiplication routine generated by the compiler apparently had some problems that significantly affect the execution times. When we link the program with the BLAS routines then we could really see serious improvements.

5. Conclusions and Future Work

On our experience with these and other algorithms ported to Itanium is that IA64 should provide a boost in performance in the order of 1.3 to 1.5 percents with just a compilation. If an algorithm ported to IA64 is not on that boundary then more aggressive optimizations are needed. At first we started trying to use compiler flags to help us in this optimization but all our efforts were unsuccessful. Our improvement occurs when we start linking our mathematical functions with the HP MLIB tools. The linking process was pretty much straight forward and no major issues were found. After that, the algorithms that took advantages of the new functions see huge improvements, but on other algorithms like FIM that did not use these routines heavily have big performance impacts.

The Itanium architecture relies most of its performance on the compiler interpretation of the code. All instructions are transformed into bundles of instructions and these bundles are processed in a parallel fashion between the different four functional units. The idea is that all functional units will be executing instructions simultaneously. But sometimes the compiler can not generate successful bundles of instructions causing ‘split issues’, meaning that functional units are stalled waiting for instructions. This issue can seriously impact the program performance and it causes programs to run slower on IA64 than on IA32. Also we need to clarify that these are very high demanding computing intensive applications that required specific architectural knowledge to fully exploit the processor capabilities.

As next steps, we plan to finish porting other routines to BLAS. We have identified some routines like matrix inverse, matrix norms, vector copies and vector maximum and minimum values that could be used on our implementations. Moreover, since we consider most of the Itanium benefits resides on its large amount of cache it will be interesting to analyze cache misses and hits on all algorithms and try to generate a correlation among the execution times.

6. Acknowledgments

This work has been supported by the Hewlett-Packard Technology Center at Puerto Rico and the NSF Engineering Research Center for Subsurface Sensing and Imaging Systems (CenSSIS).

7. References

1. P. Swain and S. Davis, *Remote Sensing: The quantitative Approach* (McGraw-Hill, New York, 1993).
2. S. Mazer and M. Martin, "*Image processing software for imaging spectrometry data analysis*", Remote Sensing of the Environment. Vol 24 no. 1. 201-210. 1988.
3. J. A. Richards and X. Jia, *Remote Sensing Digital Image Analysis* (3rd Edition, Springer-Verlag, 1999)
4. ATLAS: http://math-atlas.sourceforge.net/
5. BLAS: http://www.netlib.org/blas/
6. LAPACK: http://www.netlib.org/lapack/
7. CLAPACK: http://www.netlib.org/clapack/
8. HP MLIB: http://www.hp.com/go/mlib
9. VecLIB: http://www.nasoftware.co.uk/libraries/veclib.html
10. SCILIB: http://www.netlib.org/scilib/
11. S. Graham, P. Kessler, M. McKusick, "*gprof: A Call Graph Execution Profiler*", Proceedings of the Symposium on Compiler Construction,, Vol. 17, No 6, pp. 120-126, June 1982.