Abstract

Many signal processing algorithms can be accelerated using reconfigurable hardware. To achieve a good speedup compared to running software on a general purpose processor, fine-grained control over the bitwidth of each component in the datapath is desired. This goal can be achieved by using NU’s variable precision floating-point library. To analyze the usefulness of the floating-point divide unit, we incorporate it into our previous implementation of the K-means clustering algorithm applied to multispectral satellite images. With the lack of a floating-point divide hardware implementation, the mean updating step in each iteration of the K-means algorithm had to be moved to the host computer for calculation. The new means calculated on the host then had to be moved back to the FPGA board for the next iteration of the algorithm. This added data transfer overhead between the host and the FPGA board. In this work, we use the new fp_div module to implement the mean updating step in FPGA hardware. This greatly reduces the communication overhead between host and FPGA board and further accelerates run time. The K-means clustering example illustrates the use of the fp_div, fix2float and float2fix modules seamlessly assembled together in a real application. It is the first implementation that has the complete K-means computation done in FPGA hardware. Our results show that the hardware implementation achieves a speedup of over 2150x for core computation time and about 11x for total run time including data transfer time. They also show that the divide in FPGA hardware is 100 times faster than in software. Moreover, implementing divide in the FPGA frees the host to work on other tasks concurrently with K-means clustering, thus providing further speedup by allowing the image analyst to exploit this coarse grained parallelism.

1 Introduction

Floating-point divide is an important operation in many high performance signal processing applications including matrix inversion, vector normalization, least square lattice filters and Cholesky decomposition. However, due to the inherent complexity of division and limited resources on an FPGA, algorithms that require floating-point divide have always been carefully avoided. Alternatively, the computational part excluding floating-point divide can be accelerated on an FPGA but the floating-point divisions still have to be done on the host processor. Although many floating-point division implementations on FPGAs have been studied, most of the floating-point dividers implemented have long latency. Some use iterative algorithms and cannot be easily fit into a pipelined design. Previous publications discuss divide implementations only and do not present applications that make use of floating-point division.

We have previously presented a floating-point divide module [18] as part of the NU floating-point library. Our floating-point divide is fully pipelined, non-iterative, and has low latency. It is also small, using only a small portion of slices as well as embedded multipliers, and embedded BlockRAMs on an Xilinx Virtex-II FPGA. It allows the potential fine-grained parallelism realized by having multiple floating-point dividers as well as other logic in one FPGA. To demonstrate the practical use of the floating-point divide in real signal or image processing applications, we incorporate it into a previous implementation of the K-means clustering algorithm applied to multispectral satellite images [1] with a hybrid fixed and floating point arithmetic architecture.

The rest of this paper is organized as follows: Section 2 reviews the K-means algorithm, Section 3 presents the hardware implementation of K-means clustering, experimental results are analyzed in Section 4, and conclusions are given in Section 5. First we present related work.
1.1 Related Work

Unsupervised clustering is a classic technique for segmentation of multispectral and hyperspectral images into \( K \) clusters. K-means is a simple iterative algorithm that generates successive clustering using a computationally expensive approach. Leeser et al. [12] first proposed a hardware implementation of the K-means clustering algorithm. Manhattan norm is proposed for distance calculation to reduce the number of multipliers without sacrificing the clustering quality. The authors also proved that K-means algorithm can tolerate considerable bitwidth truncation of input data with little degradation in clustering quality [11]. This leads to a more compact and faster implementation. Expanding on their original ideas, they implemented K-means clustering of multispectral image on an Annapolis Wildstar board with three Xilinx Virtex1000 FPGAs and 40MB SRAM [5]. All input image data is pre-loaded to the onboard memory. The most computation intensive part, distance calculation, is done in the FPGA. To accommodate arbitrary input data sets and faster generation of a hardware solution, a parameterized K-means implementation was also presented [1, 10].

Other early research was carried out by Lavenier [9], who implemented K-means clustering for hyperspectral images on various reconfigurable computing engines such as Wildstar, SLAAC-1V and Spyder boards. In his work, only the distance calculation is done in the FPGA. The input image data is stored on the host and the data is streamed to the FPGA, so arbitrary size input images can be processed. This approach scales well to large input data sets but has more communication overhead between processor and hardware. Filho et al. [3] used a more accurate and expensive Euclidian distance in their hardware/software codesign K-means algorithm. The distance calculation is carried out in the Xilinx Spartan-II FPGA on a XESS PCI board. However, the hardware/software codesign only outperforms the software approach by about 2 times. To overcome the significant communication overhead between host processor and reconfigurable board due to slow I/O busses, K-means clustering on a hybrid processor was experimented with [6]. Two Altera Excalibur boards were used – one with a soft IP core 32-bit NIOS processor and one with a hard IP core ARM processor. A maximum speedup of 11.8x is achieved compared to a software implementation using the dual-port memory of the Excalibur ARM hybrid processor.

The most recent study on image data [13, 15] uses a variation of the K-means algorithm. First, a simplified filtering algorithm is used so that no more than 24 cluster centers need to be compared no matter how large the number of clusters \( K \). Second, to handle large input datasets, a simplified FEKLM algorithm is used to reduce the number of pixels that need to be scanned in every repetition. These techniques allow K-means clustering to consume much less time for large size images with large number of clusters. An average of 20-30 fps(frame per second) is achieved for a 756x512 pixel image divided into 256 clusters when implemented on an ADM-XRC-II board with one Xilinx XC2V6000 FPGA and 32MB SRAM.

Recently, clustering has been used in wider areas other than segmentation of hyperspectral images. One application in network security [8] is to cluster network traffic for anomaly detection. Another application is for document clustering [2].

The K-means clustering in this paper is an extension of our previous work [1]. It is parameterizable, highly parallel, and fully pipelined on a larger and faster Xilinx Virtex2 XC2V6000 FPGA. The novel part of this implementations of K-means clustering is the use of floating-point divide [18]. Other than the most popular digit recurrence division algorithms [16, 17, 19], we use the Taylor series based algorithm [7] that only needs a mix of small lookup tables and small multipliers. It is a non-iterative algorithm and leads to a good tradeoff of area, latency and throughput. With this fully-pipelined \( \text{fp}_\text{div} \) module, except the mean initialization at the beginning of algorithm, all computation in each iteration is now done on the FPGA, relieving the data transfer overhead between the host and the FPGA during each iteration; while in most previously published work, only the distance calculation is done in hardware. An efficient memory hierarchy is deployed in this implementation. All input images are pre-loaded from the host at the start of computation and stored in onboard DRAM during the many iterations of the computation; the clustering result during computation is stored in onboard SRAMs and sent back to the host at the end; and the new means of each iteration are stored in on-chip memory.

2 K-means Clustering

The K-means clustering algorithm is commonly used for segmentation of multi-dimensional data as shown in Fig. 1. Fig. 1(a) represents the input image data where each data pixel \( x_{ij} \) has \( C \) spectral components. Fig. 1(b) is the output clustered image. In this example, the input image spectral data is classified into 5 different classes. In previous work [1] we applied this algorithm to multispectral satellite images using reconfigurable hardware. K-means works by assigning multidimensional vectors to one of \( K \) clusters, where \( K \) is given a priori. The aim of the algorithm is to minimize the variance of the vectors assigned to each cluster. The algorithm is iterative: after each vector is assigned to one of the clusters, the cluster centers are recalculated and the vectors are re-assigned using the new cluster centers.

Pseudo-code describing the K-means algorithm is given in Fig. 2. For more details see Duda and Hart [4]. In the pseudo-code, a total of \( N \) pixels are processed and assigned
to one of the $K$ clusters. The variable centers is an array of $K$ multidimensional vectors which represent the center point or mean for each cluster. Each cluster has a counter and an accumulator for pixel values assigned to it. The algorithm works as follows. First, means are initialized; we use a sub-sampling method. Next the while loop iterates until a termination condition is met. During each iteration, the distance to each cluster center is calculated for each pixel in the input image. The shortest distance is kept and the pixel is assigned to the closest cluster based on the distance between each pixel and each of the $K$ cluster centers. Each pixel is then added to the accumulator of the cluster it is assigned to and the corresponding counter is incremented by one. After accumulation, the new mean of each cluster is calculated and is used for the next iteration. The new mean is obtained by dividing the accumulator value by the counter value. In previous work [1] this mean updating step was done on the host because it requires floating-point divide. With our new $fp$ div module, we are able to implement mean updating in FPGA hardware. This eliminates the overhead of moving the counter and accumulator values for each cluster from FPGA to host and moving the new means – the result of accumulator divided by counter for each cluster – from host back to the FPGA for each iteration. The use of the floating-point divide in FPGA hardware further accelerates the run time of the K-means algorithm. At the end of each iteration, the termination condition is evaluated and, if not met, the process is repeated. Possible termination conditions are convergence or processing for a fixed number of iterations. When done, the algorithm returns the clustered image as well as the final mean values of all clusters.
3 Design Implementation

3.1 Reconfigurable Hardware Platform

Many reconfigurable computing systems are based on one or more FPGAs connected to a number of memory banks. All designs presented in this work are implemented on one such architecture, a VantageRT FCN board from Mercury Computer Systems, Inc. [14]. The block diagram of this board is shown in Fig. 3. The board integrates one Xilinx Virtex-II XC2V6000 FPGA with two 500 MHz PowerPC G4 microprocessors, 12MB of DDR SRAM and 256MB of DDR SDRAM both running at 133MHz. In this work, we did not make use of the PowerPCs on this board.

![Figure 3. Architecture of the Mercury reconfigurable computing board](image)

3.2 High Level Design Structure

We implemented the K-means clustering algorithm from Fig. 2 on the Mercury VantageRT FCN board shown in Fig. 3. In this work, we use the Xilinx Virtex-II XC2V6000 FPGA, DRAMs and SRAMs on the board. The implementation is partitioned between the host and the reconfigurable hardware. Cluster means are initialized on the host while the rest of the algorithm is done in reconfigurable hardware. Fig. 4 shows the interaction between the host and the reconfigurable hardware. All communication from host to reconfigurable board is over the PCI bus. The operation of K-means clustering proceeds as follows. First, all pixels of the input spectral image are moved from the host to the DRAMs on the FPGA board. Then, initial cluster means are moved from host to the FPGA on the reconfigurable board. The input data pixels are stored in DRAMs on the board during the entire computation and are not changed during the iterations. The new means after each iteration are stored in the on-chip memory of the FPGA. This is efficient because reading/writing of FPGA on-chip memory only requires one clock cycle while reading/writing of onboard memory can take many clock cycles. Due to the limited on-chip memory, the cluster assignment of each pixel is stored in onboard SRAM during computation. After the algorithm terminates, the final means stored in on-chip memory are moved back from the FPGA to the host, and the final cluster assignments are moved back to the host from onboard SRAM.

![Figure 4. Interaction between the host and the reconfigurable hardware](image)

The reconfigurable hardware portion of the K-means clustering algorithm consists of three basic functions: assigning each pixel in the input image to one of the \(K\) clusters, accumulating the values of all pixels that belong to every cluster, and updating the mean for each cluster. These operations are performed on each pixel in the input image serially. The pixels are streamed from onboard memory and processed one per clock cycle until all pixels are processed. Fig. 5(a) shows the overall K-means clustering circuit. It is composed of three functional units as well as two shift-register units, validity and pixel shift that synchronize the pixel data with the computation. Inputs to the circuit are all pixels in the input image and the initial means of all \(K\) clusters; outputs are the cluster assignment for each pixel and the final value of the means of all \(K\) clusters.

One functional unit is the datapath unit shown in Fig. 5(b). It takes one pixel datum at each clock cycle along with all the cluster centers as inputs, and outputs the cluster assignment for that pixel. Each pixel is assigned to its nearest cluster, which is obtained by comparing the Manhattan(1-norm) distance between that pixel and every cluster center. This is the most computationally intensive part of the implementation. Another functional unit is the accumulator. Each accumulator is affiliated with one cluster and accumulates the total value of all pixels that belong to that cluster. There is also a counter associated with each cluster used to count the total number of pixels that belong to the cluster. The third functional unit is the floating-point divider for mean updating shown in Fig. 5(c). Two shift units are used to delay signals from memory to the accumulator, thus synchronizing them with the cluster assignment which is delayed due to the latency of the datapath.
Figure 5. Functional units of the K-means circuit

unit. The pixel shift unit is a set of shift registers used to pass pixel values from memory to the accumulator in parallel with their operations in the datapath unit. The validity unit sets a Data Valid signal that indicates the pixel data is valid and enables the accumulator.

In our implementation, both datapath and accumulator units are implemented in fixed-point arithmetic while the divide unit is floating-point for improved precision. Converters from fixed to floating-point representation are placed before the floating-point dividers while floating to fixed-point converters are placed after the dividers. This hybrid implementation demonstrates that our variable precision library can support hybrid fixed and floating-point implementations.

### 3.2.1 Datapath Unit Complexity

The datapath unit determines the cluster assignment for each pixel. For each pixel, the cluster it belongs to is determined by first computing the distance of that pixel to every cluster center, using the Manhattan or 1-norm distance. The shortest distance is then found by comparing all distances. The pixel is assigned to its nearest cluster to minimize the variance within each cluster. For multispectral images, each pixel has several dimensions (or channels). Thus, the distance calculation needs to be done for each channel and the result is added for each pixel and cluster center pair. Assume that a $C$ dimensional input image is segmented into $K$ clusters. The Manhattan distance between one input pixel $\text{pixel}_i$ and one cluster center $\text{cluster}_k$ is obtained by:

$$\sum_{c=1}^{C} |\text{PIXEL}_i(c) - \text{CLUSTER}_k(c)|$$  \hspace{1cm} (1)

We compute the Manhattan distance calculation for all $K$ clusters in parallel. This calculation requires $K \times C$ subtractions, $K \times C$ absolute values, and $K \times (C - 1)$ additions. For an image with 8 channels and 8 clusters, 64 subtractions, 64 absolute values, and 56 additions are required. An adder tree with $\sum_{i=1}^{\log_2 C} C/2^i$ adders is used to sum up absolute distances of all $C$ channels for each cluster. Altogether 8 adder trees with a total of 7 adders each are required. Once the distances between one pixel and all $K$ cluster centers are known, $K$ comparators are required to find the shortest distance. Again, a comparison tree is adopted to exploit more parallelism. In this example, we need one comparison tree of 7 comparators. In summary, a total of 191 operations are needed.

### 3.2.2 Mean Update Unit

After the datapath unit computes the cluster assignment of each pixel, the accumulator unit collects the total number of pixels assigned to this cluster, stored in register $\text{counter}$; and the total value of all pixels associated with this cluster, stored in register $\text{accumulator}$. New mean values for each cluster are calculated by dividing the accumulator value by the $\text{counter}$ value. Note that there are $K$ counters and $K$ accumulators, one pair for each cluster. In this
work, the mean update division is done in FPGA hardware. For arithmetic accuracy, divide is implemented in floating-point using the \texttt{fp\_div} module. Since both the counter and accumulator are in fixed-point format, \texttt{fix2float} conversion has to be done first. After the floating-point divide, the new mean in floating-point format needs to be converted back to fixed-point format using the \texttt{float2fix} module. For an image with 8 clusters, a total of 8 \texttt{fp\_div} modules are in parallel, one for each cluster. Also, we need 8 \texttt{float2fix} modules, and 16 \texttt{fix2float} modules (8 for counter and 8 for accumulator).

The floating-point divide [18] and float to fixed conversion are modules in the Northeastern University Reconfigurable Computing Laboratory variable precision floating-point library\textsuperscript{1}. The floating-point divide implemented is based on Taylor series approximation [7]. This algorithm can be efficiently implemented on a FPGA and makes use of embedded RAM and multipliers. Assume a dividend \(X\) and a divisor \(Y\) both in the range of \([1,2)\). The divisor \(Y\) is decomposed into a higher order \(m\)-bit part \(Y_h\) and a lower order \(m\)-bit part \(Y_l\) so that \(Y_h > 2^m Y_l\). Using Taylor series expansion:

\[
\frac{X}{Y} = \frac{X}{Y_h + Y_l} = \frac{X}{Y_h} \left(1 - \frac{Y_l}{Y_h} + \frac{Y_l^2}{Y_h^2} - \ldots \right) \approx X \times (Y_h - Y_l) \times \frac{1}{Y_h^2}
\]

(2)

Fig. 6 shows a block diagram of equation (2). The Block-RAMs and embedded multipliers available on the Xilinx Virtex-II FPGA are used in our implementation for table lookup and multiplication.

4 Experimental Results

We implemented K-means clustering on the Mercury VantageRT FCN board for a multispectral satellite image with \(614 \times 512\) pixels. Each pixel has 8 channels, with 8 bits of data per channel. The input pixels are segmented into 8 clusters. One resulting clustered image, shown in Fig. 7, uses eight different colors to represent the different clusters. Note that this is a “pseudo-color image” with colors that carry no information randomly assigned to clusters.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure6.png}
\caption{Table-lookup based divider}
\end{figure}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Number of iterations & Core computation time (seconds) \tabularnewline
\hline
1 & 156 \tabularnewline
2 & 312 \tabularnewline
4 & 624 \tabularnewline
8 & 1248 \tabularnewline
\hline
\end{tabular}
\caption{Run time comparison of pure software and hardware implementation}
\end{table}

\textbf{Figure 7. Output clustered image: the city of Denver}

Our design is written in VHDL and synthesized using Synplify Pro 8.0. The bitstream is generated using Xilinx ISE 6.3i and downloaded to the XC2V6000 FPGA on the board. All the results that we present in this section are based on hardware running on the VantageRT FCN board.

To verify the correctness of the K-means clustering algorithm implemented in the FPGA, we compared it against a pure software implementation on a 3.2GHz Intel Pentium 4 processor. Both implementations generate the same results of final mean and cluster assignment for each pixel. To see the advantage of having floating-point divide in FPGA hardware instead of running mean update on the host, we experimented with two implementations, one with divide in the FPGA and the other with divide on the host. In the following discussion, we compare the results of these three implementations: pure software, hardware implementation with divide in FPGA hardware, and hardware implementation with divide on the host.

4.1 Hardware vs. Software

Table 1 shows the run time of the pure software implementation and hardware implementation with divide in the FPGA for different numbers of iterations. The first column is the number of iterations that the algorithm runs. The second and third columns are the core computation time and total run time (both in seconds) of the pure software implementation. The fourth and sixth columns are the core computation time and total run time of the hardware implemen-
4.1 Divide in the FPGA vs. Divide on the Host

The run time comparison of the two hardware implementations, with divide in the FPGA and with divide on the host, is described in Table 2. Similar to Table 1, it lists the core computation time and total run time for different numbers of iterations. Table 2 also shows the time for “divide” only, which is part of the core computation.

One interesting observation is that the difference between these two implementations is fairly small. This is not hard to explain with Amdahl’s Law. We can see from Table 2 that “divide” takes a very small fraction of the core computation time, and even a smaller portion of the total run time. The run time is dominated by the fixed point datapath. For the first implementation with divide on the host, in one iteration “divide” takes 0.1 ms, which is about 4% of the core computation time while the core computation time is only about 0.009% of the total run time. After moving “divide” from the host to the FPGA, the time spent for divide is accelerated 100 times, to about 1 µs. Now the fraction of time spent on divide in the core computation is even smaller—only 0.04%. Since the “divide” is such a small portion of the whole computation, no matter how significant the improvement is, the effect on the whole design is modest.

The main advantage of having the “divide” in the FPGA is to free the host to work on other tasks concurrently with K-means being implemented on the FPGA board. In scenarios where an image analyst is using K-means as a pre-processing step, further processing can be done on the host while clustering is performed on another image. The ability to exploit this coarse grained parallelism will provide significant speedup for the image analyst.

Table 1. Run time of K-means Clustering Algorithm: Hardware vs. Software

<table>
<thead>
<tr>
<th># of iterations</th>
<th>Software</th>
<th>Hardware (divide in FPGA)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>comp(s)</td>
<td>total(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>5.203</td>
<td>38.047</td>
</tr>
<tr>
<td>20</td>
<td>104</td>
<td>137</td>
</tr>
<tr>
<td>50</td>
<td>258</td>
<td>291</td>
</tr>
<tr>
<td>1000</td>
<td>5188</td>
<td>5220</td>
</tr>
<tr>
<td></td>
<td>0.0024</td>
<td>2168</td>
</tr>
<tr>
<td></td>
<td>0.048</td>
<td>2167</td>
</tr>
<tr>
<td></td>
<td>0.12</td>
<td>2150</td>
</tr>
<tr>
<td></td>
<td>2.4</td>
<td>2162</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>total(s)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td></td>
<td>30</td>
<td>174</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>total speedup</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.4</td>
<td>5.1</td>
</tr>
<tr>
<td></td>
<td>10.8</td>
<td></td>
</tr>
</tbody>
</table>

4.2 Divide in the FPGA vs. Divide on the Host

The run time comparison of the two hardware implementations, with divide in the FPGA and with divide on the host, is described in Table 2. Similar to Table 1, it lists the core computation time and total run time for different numbers of iterations. Table 2 also shows the time for “divide” only, which is part of the core computation.

One interesting observation is that the difference between these two implementations is fairly small. This is not hard to explain with Amdahl’s Law. We can see from Table 2 that “divide” takes a very small fraction of the core computation time, and even a smaller portion of the total run time. The run time is dominated by the fixed point datapath. For the first implementation with divide on the host, in one iteration “divide” takes 0.1 ms, which is about 4% of the core computation time while the core computation time is only about 0.009% of the total run time. After moving “divide” from the host to the FPGA, the time spent for divide is accelerated 100 times, to about 1 µs. Now the fraction of time spent on divide in the core computation is even smaller—only 0.04%. Since the “divide” is such a small portion of the whole computation, no matter how significant the improvement is, the effect on the whole design is modest.

The main advantage of having the “divide” in the FPGA is to free the host to work on other tasks concurrently with K-means being implemented on the FPGA board. In scenarios where an image analyst is using K-means as a pre-processing step, further processing can be done on the host while clustering is performed on another image. The ability to exploit this coarse grained parallelism will provide significant speedup for the image analyst.

5 Conclusions

An implementation of the K-means clustering algorithm for multispectral satellite image processing is presented in
Table 2. Run time of Two Hardware Implementations of K-means Clustering Algorithm

<table>
<thead>
<tr>
<th># of iterations</th>
<th>Hardware(divide on host)</th>
<th>Hardware(divide in FPGA)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>comp(s)</td>
<td>divide(s)</td>
</tr>
<tr>
<td>1</td>
<td>0.0025</td>
<td>0.0001</td>
</tr>
<tr>
<td>20</td>
<td>0.05</td>
<td>0.002</td>
</tr>
<tr>
<td>50</td>
<td>0.125</td>
<td>0.005</td>
</tr>
<tr>
<td>1000</td>
<td>2.5</td>
<td>0.1</td>
</tr>
</tbody>
</table>

this paper, demonstrating the use of the floating-point divide from the NU floating-point library. Speedup is shown for moving the mean updating step from host PC to FPGA hardware with the \texttt{fp\_div} module, eliminating the data transfer overhead between the host and the FPGA for each iteration. We also see that the data transfer time in all implementations dominates the run time. This includes the time for reading the image data from an external input file, moving input pixel data from the host to the DRAM on the board, and the time for moving final cluster assignments stored in SRAM on the board, as well as the final means stored in on-chip memory on the FPGA back to the host and writing results to an output file. Since divide is a small percentage of the K-means algorithm, Amdahl’s Law tells us that speeding up division will not have a great impact on overall run time. The great benefit comes from being able to exploit the coarse grained parallelism of running different applications on the host and the FPGA concurrently.

Acknowledgement

We would like to thank researchers at Los Alamos National Laboratory for their early work on this project. We also would like to thank Mercury Computer Systems, Inc. for providing the VantageRT FCN board. This research was funded in part by Mercury Computer Systems, Inc. and in part by CenSSIS, the Center for Subsurface Sensing and Imaging Systems, under the Engineering Research Centers Program of the National Science Foundation (award number EEC-9986821).

References


