Architectural Challenges in Memory-Intensive, Real-Time Image Forming

A. Åhlander[1], H. Hellsten[2], K. Lind[2], J. Lindgren[2], and B. Svensson[1]
1. Centre for Research on Embedded Systems, Halmstad University, SE-301 18 Halmstad, Sweden
2. Saab Microwave Systems, SE-412 89 Gothenburg, Sweden
Email: anders.ahlander@ide.hh.se (corresponding author)

Abstract

The real-time image forming in future, high-end synthetic aperture radar systems is an example of an application that puts new demands on computer architectures. The initial question is whether it is at all possible to meet the demands with state-of-the-art technology or foreseeable new technology. It is therefore crucial to understand the computational flow, with its associated memory, bandwidth and processing demands. In this paper we analyse the application in order to, primarily, understand the algorithms and identify the challenges they present on a basic architectural level. The processing in the radar system is characterized by working on huge data sets, having complex memory access patterns, and doing real-time compensations for flight path errors. We propose algorithm solutions and execution schemes in interplay with a two-level (coarse-grain/fine-grain) system parallelization approach, and we provide approximate models on which the demands are quantified. In particular, we consider the choice of method for the performance-intensive data interpolations. This choice presents a trade-off problem between computational performance and size of working memory. The results of this "upstream" study will serve as a basis for further, more detailed architecture studies.

1. Introduction

A challenge for architects of embedded computer systems is the processing in future, advanced image creating sensor systems, where complex transformations of massive sensor data sets are carried out. An example is the signal processing in low-frequency synthetic aperture radar (SAR) systems. The SAR system investigated in this study is an ultra-wideband system which operates in the low VHF-band (20-90 MHz). The low frequencies give, e.g., efficient foliage penetration and biomass estimation. The ultra-wideband technology implies for SAR a resolution that approaches the wavelength limit. As a comparison, the resolution in a traditional microwave SAR system typically equals several wavelengths.

The SAR technique was developed in the early 1950s and is used to create high-resolution radar images from low-resolution aperture data. A multitude of image formation algorithms have been developed [1]. The algorithms have primarily operated in the frequency domain. These are computationally efficient but have the shortcoming that they are derived for a linear aperture (flight track) which is an assumption that is not valid for an ultra-wideband system. The image formation can instead be done in the time-domain with back-projection techniques similar to those in computed tomography (CT) [2]. One advantage with the time-domain processing is that it is possible to compensate for non-linear flight tracks, however typically to the cost of lower computational efficiency. There is an important difference between SAR and CT. In SAR the sensor detections are averages over arcs of responses, whereas in CT the detections are averages over straight lines of responses. This leads to more complex data index calculations in the SAR back-projection than in the CT back-projection. Moreover, compensation for sensor path deviations has to be done in the SAR processing.

The Fast factorized back-projection (FFBP) [3] [4] is a computationally efficient algorithm for image forming in the time domain. It exploits a fundamental redundancy in radar data and reduces the performance requirements substantially relative to those for traditional global back-projection (GBP) techniques. The FFBP has been well studied from an algorithm performance point of view, and has shown good imaging performance as well as potential for efficient execution, well in parity with the FFT-based algorithms in frequency domain processing.

Even though the FFBP has been implemented in software, and tested on radar data, it has not yet been thoroughly investigated from a real-time computer realization point of view. The image calculations must keep

2007 International Conference on Parallel Processing (ICPP 2007)
0-7695-2933-X/07 $25.00 © 2007 IEEE

up with the ground coverage of the radar. I addition, the latency must be as low as possible. The calculations include compensations for flight path deviations. A number of computational challenges can be identified: In these SAR systems the integration time is typically several minutes during which huge amounts of data are collected. In the larger systems a working memory of hundreds of gigabytes will be needed. The computational performance demands will be tens or hundreds of GFLOPS. The large data sets represent themselves a challenge but also the complicated memory addressing scheme. Interpolation kernels are swept along curved paths in the memory space during the execution. Due to changing geometry proportions, the shapes and positions of these paths vary as the image forming progresses. In addition, the exact properties of the curves are not known a priori, because these are dependent on compensations for non-linear flight paths and thus must be extracted on the fly. These compensations require geometrical calculations, which need to be intermixed with the actual image calculations. The locality of references is another aspect; different read paths that contribute to the same outdata may be far away from each other in the memory space. The calculation demands can also be considerable, depending on, e.g., the chosen interpolation methods.

The need for efficiency in the FFBP execution is emphasized by the fact that the embedded processing has tight power and size requirements. The processor architecture should also be scalable in order to suit a platform span from small, one-engine aircrafts to large surveillance systems. Since the calculations are demanding, both from the performance and from the memory point of view, a parallel computer architecture is needed. A major difference in this application compared to another well-known demanding radar application, the space-time adaptive processing (STAP) [5][6], is seen in the relation between the performance demands and the memory size. In STAP the performance demands are huge while the memory demands are moderate; in SAR it is the other way around. Thus the number of operations per data element differs significantly which, e.g., means that the requirements on the memory subsystem are very different.

In this paper we analyse the algorithms with regard to the fundamental principles for parallelization. The analysis can serve as a basis for further, more detailed architecture studies for the various platform types. The paper is organized as follows. The next section gives a brief introduction to time-domain SAR image forming, including GBP and FFBP. Section 3 discusses overall solutions for efficient parallelization and execution of the FFBP. Section 4 deals with different design choices for data interpolations, which are shown to have a great impact on the overall system. Section 5 focuses on solutions for efficient execution of the interpolations. Finally, Section 6 draws the main conclusions of the study.

2. Time-domain SAR image forming

A SAR system produces (in real time) a high-resolution map of the ground while the platform is flying past it. In SAR the radar transmits a relatively wide beam to the ground, illuminating each resolution cell over a long period of time. Figure 1 illustrates a SAR system that creates a map over a continuous ground strip; the system works in a *stripmap mode*. The system continuously processes incoming data and produces image data. The integration angle is about 90 degrees. The integration length may be tens of kilometers, which corresponds to an integration time of several minutes.

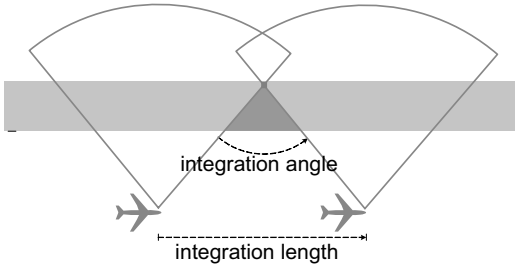


Figure 1. Stripmap processing. The integration length may be tens of kilometers.

The effect of this movement is that the distance between a point on the ground and the antenna varies over the data collection interval. This variation in distance is unique for each point in the area of interest.

Figure 2 illustrates the radar data collection when *M*x*N* resolution cells are illuminated. The radar transmits pulses which are repeated with a certain *pulse repetition frequency*. For each transmitted pulse, *M* samples, or *range bins*, are collected. Each range bin corresponds to a particular distance to the ground cell, as the distance is proportional to the time until the echo is received. Sampled echoes from *L* radar pulses are placed in a radar data matrix. A row in the matrix corresponds to a particular pulse and thus to a platform position along the flight path. Likewise, a column corresponds to a particular target range. I should be noted that the flight track not always can be assumed linear.

2007 International Conference on Parallel Processing (ICPP 2007)
0-7695-2933-X/07 $25.00 © 2007 IEEE

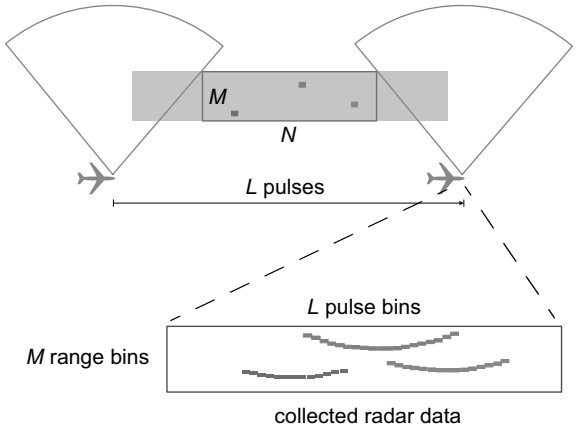


Figure 2. Collecting radar data. *M*x*N* resolution cells are illuminated.

The task for the signal processor is to integrate, for each resolution cell in the output image, the instantaneous response that a target in that particular cell would have. This is illustrated in Figure 3, where the created image is of the size *M*x*N* pixels. There are several algorithms that can be used to compute this integral [1]; the choice is a trade-off between accuracy and computational load. The images can be calculated with FFT techniques in the frequency domain, or with back-projection techniques in the time domain. Time domain techniques are described below.

2.1 Global back-projection

In Figure 3 the global back-projection is illustrated. L radar pulses correspond to the full radar integration length, or *aperture*. Each of the MN pixels in the resulting image is created by L calculations (additions). Thus the computational complexity is proportional to LMN.

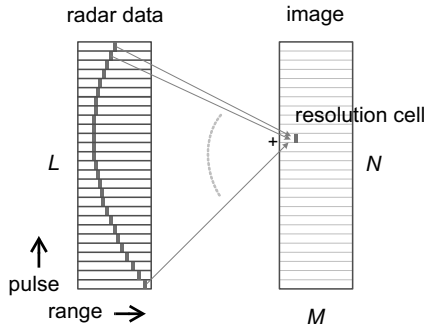


Figure 3. Simplified illustration of global back-projection. Each image resolution cell corresponds to a unique curve in the radar data.

2.2 Fast factorized back-projection

The FFBP is a computationally efficient algorithm for image forming in the time domain. It exploits redundancy in radar data and reduces the performance requirements substantially relative to the demands from traditional GBP. Initially, the whole aperture with length L consists of a large number of small *subapertures* with low angular resolution. These subapertures are iteratively merged into larger ones with higher angular resolution, until the full aperture with full angular resolution is obtained. In each iteration k, sets of n subapertures of length l_k are combined into subapertures of length nl_k. Each of the MN resulting pixels in one iteration is created by n calculations. There are $\log_n L$ iterations. This sums up to $nMN \log_n L$ calculations. The integer value of n that gives the lowest operation count is 3 (the actual value is e, the base for natural logarithms). If this value is chosen, the ratio between global and factorized back-projection operation counts becomes $L/3\log_3 L$. If the length of the aperture is 10^5 this ratio equals 3181. Thus a substantial alleviation of the computational burden is achieved. Figure 4 illustrates in a very simplified way how a resolution cell is created iteratively from the radar data. Here n equals 3. In each iteration, a data element in a resulting aperture is created from three data elements, one from each contributing subaperture.

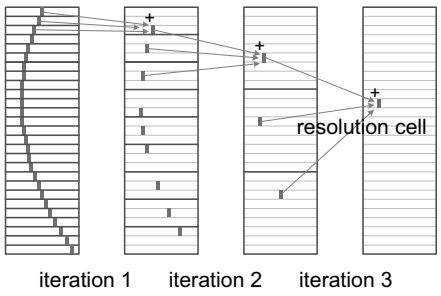


Figure 4. Simplified illustration of factorized back-projection.

3. Parallelization of the fast factorized back-projection

The described algorithms have been implemented in program code which has calculated images from real radar data. This processing has so far been done "off-line". The succeeding step is to find a computer architecture that is capable of creating the images in real time. Several processors need to share the load due to the high performance demands. The parallel processing is preferably performed on different levels of granularity. On a coarse level the whole data set is split over a number of (loosely coupled) processing nodes, each taking care of a subset of the data. The data partitioning on this level must, e.g., consider that data dependencies preferably should be kept within a data subset. On a fine granularity level the processing within a subset can

2007 International Conference on Parallel Processing (ICPP 2007)
0-7695-2933-X/07 $25.00 © 2007 IEEE

be carried out by exploiting instruction level, thread or data parallelism. Here, such a two-level parallelism is assumed.

As an illustration for the further discussions, the straightforward data representation shown in Figure 5 is assumed. It is a polar representation where each direction from the aperture is represented by a row in a data matrix. The depicted linear ground strip is then represented by a curved shape in the matrix.

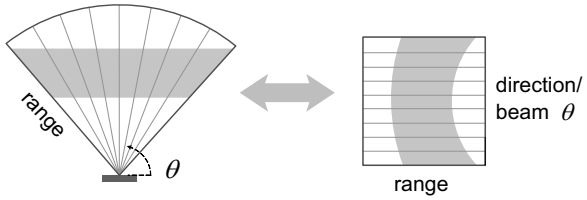


Figure 5. Polar representation of a ground strip.

3.1 Coarse-grain parallelization

It is possible to do a data partitioning as in Figure 6, where the image is divided into data independent slices along the length (azimuth) axis. However, since the response from a resolution cell migrates over several range bins over the full aperture, the slices must be overlapped to be fully data independent. We must thus have some redundancy in data. A multicomputer is the choice to prefer in this case. The processor nodes can then take care of one slide each. The system can operate with minimum communication between the nodes, thus not requiring a shared memory.

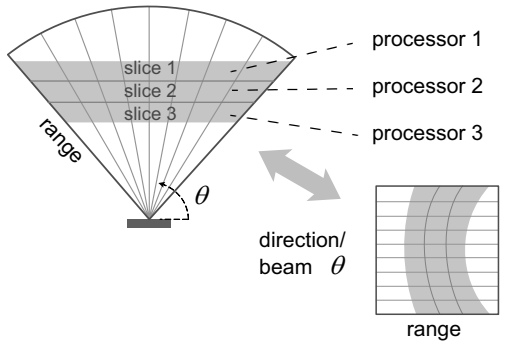


Figure 6. Coarse-grain parallelization. A number of slices with an overlap.

This solution also gives a natural system scalability; a varying number of identical nodes are used depending on the system size.

3.2 Fine-grain parallelization

The parallelization on the fine grain level is more complex than the coarse grain parallelization. A critical issue in the image forming calculations is the execution of the hot-loops, where contributing data elements are combined to new data elements. The actual data combination is basically only a couple of complex additions, but it has to be preceded by complicated index calculations and data interpolations, which generate the real performance demands. However, the index calculations can be simplified by approximations and the data interpolations can be carried out with different levels of ambition. The chosen interpolation method affects the total performance and memory demands significantly. Data interpolations are further discussed in Section 4.

3.2.1 Data locality

The memory references get increasingly scattered over the iterations, as can be seen in Figure 7 a). However, as Figure 7 b) indicates, there is a potential for good spatial locality in the references if a proper execution scheme is used. When a new image point is to be created, data is fetched from a limited region in each contributing subaperture. The size of the region depends on the chosen interpolation kernel, and the exact position depends on the degree of flight path compensation. The regions slide in the memory as the data rows in the resulting aperture are created, meaning that data can be cached and reused. Also calculations can be reused.

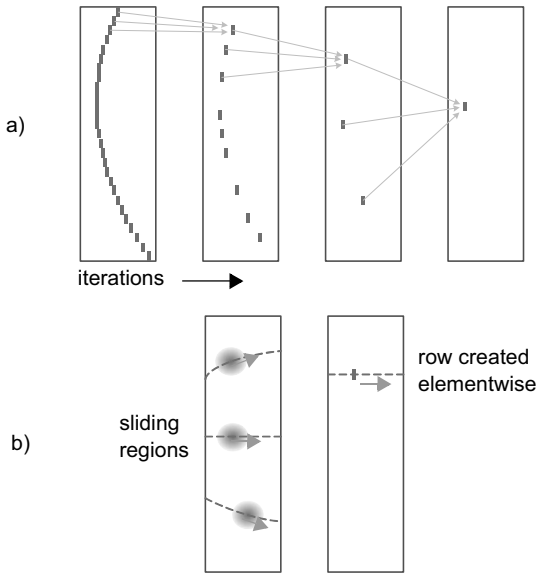


Figure 7. a) Increasing distance between contributing elements over the iterations. b) Locality in sliding windows in contributing data.

The execution, i.e. the interpolations etc., within these contributing regions, constitutes the overwhelming part of the total computation. Therefore it is extremely important that this so called hot-loop execution is carried out with high efficiency.

2007 International Conference on Parallel Processing (ICPP 2007)
0-7695-2933-X/07 $25.00 © 2007 IEEE

3.2.2 Addressing pattern and index calculations

In the FFBP iterations, the core operation is the creation of data elements in the resulting, merged aperture. The elements are created from data in the contributing subapertures. However, to create a resulting data row, data must be fetched from the contributing apertures along non-linear paths, see Figure 8. These paths get increasingly distorted over the iterations, and the indices for pointing them out are obtained by complex geometrical operations. The contributing data elements are typically calculated through interpolations in the indata set. Thus, the FFBP execution to a large extent consists of sweeping interpolation kernels along curved paths in memory. The interpolation kernels could be of various complexity with different memory and performance demands. Interpolations are further discussed in Section 4.

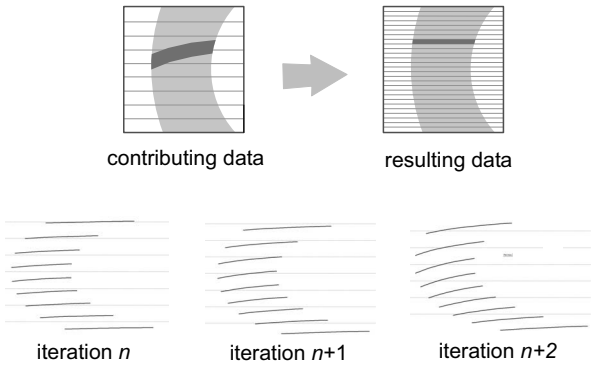


Figure 8. Data dependencies in an FFBP iteration. Each direction, or data line, in the resulting aperture data gets contributions from non-linear paths in the contributing subapertures' data.

Figure 9 a) illustrates how an interpolation kernel is swept along non-linear paths in the indata matrix while calculating the elements in the outdata matrix. The resulting data is stored in the outmatrix along the range lines.

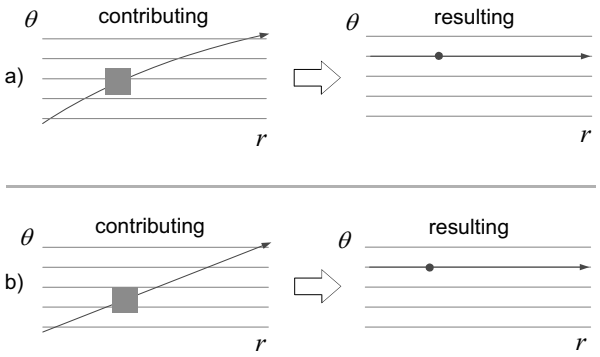


Figure 9. a) Curved read path in indata matrix. b) Read path approximated to linear.

Determining the exact shape of the read path on the fly requires much computational power, but approximations can be made. This is illustrated in Figure 9 b) where the path is approximated with a linear one. One way of doing the approximation is shown in Figure 10. For the contributing subaperture i, fixed index increments, Δr_i and $\Delta\theta_i$, are used for addressing the data matrix within the range interval of approximation, δr_{limit}.

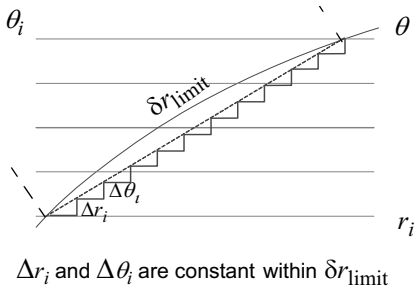


Figure 10. Approximation of memory read path.

The memory is segmented on the basis of the linearization intervals which in turn are dependent on the system requirements. This method gives a drastic reduction in the amount of geometric calculations, since they only need to be done for each interval of approximation, δr_{limit}. Memory segmentation is further discussed in Section 5.

3.3 Summary

On a coarse-grain level, a memory partitioning approach that suits multicomputers has been proposed. It exploits the inherent data parallelism in a natural way and supports a straightforward system scalability. On the fine-grain level, memory addressing and locality have been discussed. Approximations of the complicated reading paths are possible and there is good data locality in the problem. Finding ways to efficiently execute the intensive hot-loops has been identified as the greatest challenge, and this is further discussed in the continuation of this paper.

4. Interpolation methods

The method used in the data interpolations, which are carried out in the performance consuming hot-loops, is an example of a design choice that has a large impact on the whole system architecture. Figure 11 illustrates how the requirements on computational performance and memory size, in a radar system with a given system performance (resolution etc.), varies depending on the chosen method for interpolation. Basically, to get a good data estimation, an advanced and precise interpolation method does not need as high

degree of oversampling as a simpler one. Hence, the total memory requirement typically gets smaller when the interpolation gets more advanced. However, since the computational complexity in the interpolations grows fast with the size of the interpolation kernel, the total computational requirements increase, even though the amount of data is reduced.

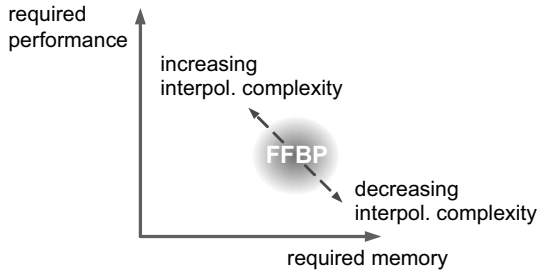


Figure 11. The memory/performance ratio of the signal processing varies with the interpolation complexity.

The above indicates that there is a tight relationship between the choices concerning the calculations on the lowest level in the hot-loops and the overall system design considerations. The overall power demand is also of vital interest here; processors for executing hot-loops are power consuming, but so are large memories. More information about interpolation methods in FFBP is given in [7] which reports studies of the image quality when using cubic and nearest neighbor interpolations. In the continuation of the report we will further study different interpolation methods, from a computational efficiency point of view. Section 5 will show sample figures for the cubic and nearest neighbor interpolations methods.

5. Analysis of fine grain parallelization

5.1 Memory segmentation

Based on the linearization intervals discussed in Section 3.2, the data area is divided into segments. Within the segments the curved read paths are approximated to parallel, linear ones, see Figure 12. The figure shows two different scenarios; a) one representing an early FFBP iteration and, b) one representing a late iteration. The memory is read in horizontal lines in the early iteration, and in tilted lines in the late iteration.

An out-data matrix is calculated by applying a filter kernel along the different linear paths indicated in Figure 12. The kernel could be of various sizes and shapes. Typical filter kernels are of size 1x1 and 4x4. Here we assume a cubic interpolation kernel (4x4) and a segment size of *S*x*S*. With an interpolation kernel of size *I*x*I* there must be an overlap of *I*/2 into the surrounding segments.

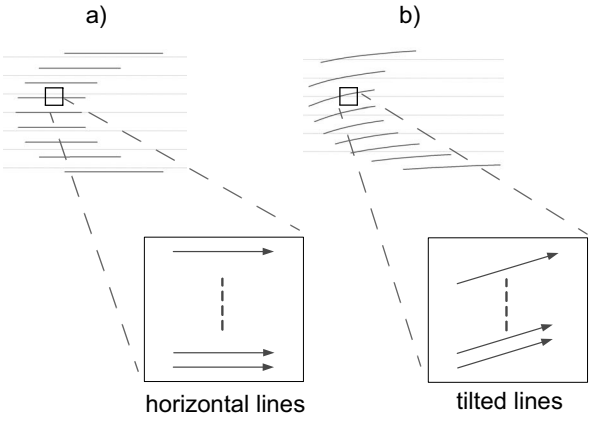


Figure 12. Memory read scenarios in two different FFBP iterations.

Figure 13 shows how a block, including the overlap into the neighbor regions, is stored in a linear memory. One sample is assumed to equal one memory word in this case. Ideally, the data is read once and buffered. Each data point is thereby only read once from memory. However, the light shaded data has to be read at least twice.

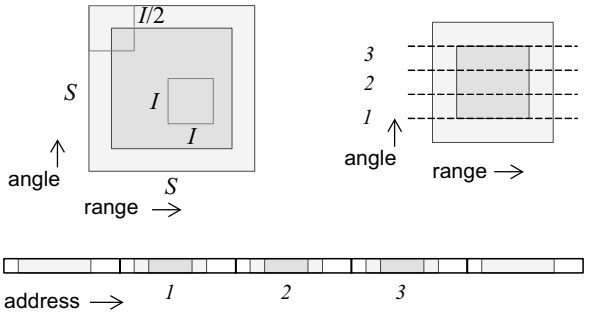


Figure 13. A segment including overlap stored in a linear memory space.

We study the potential for fine-grain parallelization, the degree of data reusability when the kernel slides, as well as the estimated buffer size for the reuse. Other important aspects are the number of operations and the number of memory accesses per interpolated data point.

5.2 Calculations in an FFBP iteration

In each FFBP iteration, the resulting data matrix is created segment by segment. This is illustrated in Figure 14, where three subapertures are combined into one (note that the segments may be overlapped, depending on the chosen interpolation kernel). Each data line in a resulting segment is created by adding the corresponding contributing data lines from the source

segments. Each of the contributing data lines is acquired by sweeping an interpolation kernel along a line in the source segment, as described in Section 5.1.

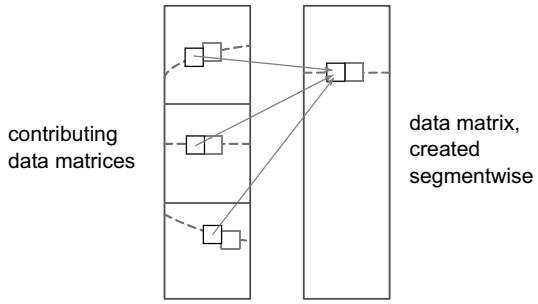


Figure 14. Resulting data matrix is created segmentwise in an FFBP iteration.

In the following, we look deeper into the data line calculations, with focus on getting efficient sweeping of interpolation kernels along lines within a segment in the contributing data matrix.

5.3 Efficient execution of sliding interpolation kernel

As the performance demands to the largest extent stem from the interpolations in the hot-loops, it is important that these are done efficiently. It is possible to reuse data read from memory in order to reduce the memory bandwidth, as well as to reuse calculation results in order to reduce the performance demands. Figure 15 a) illustrates how an interpolated value (a square in the figure) is created from samples in the contributing data set (circles in the figure) by a two-dimensional cubic interpolation. The interpolation is carried out in two steps: Intermediate, range-interpolated values are first calculated. These intermediate values (indicated by crosses) are then used to calculate the final value by an interpolation in the beam direction. Figure 15 b) indicates how intermediate data points can be reused. In the model, the resulting values are aligned in range, which means that one intermediate, range-interpolated value can be used in the calculation of several, typically four, resulting values.

The data is assumed to reside in a synchronous DRAM, which means that data should be read and written in bursts, in order to minimize the total memory latency. Figure 16 shows an execution scheme that supports burst memory reads and writes, as well as reuse of calculations and data. The figure shows the calculation of contributing data points from *one* subaperture. Data is read continuously from consecutive positions (over a segment row) in the memory, and fed into the calculation stage, which operates in a systolic fashion. The calculation stage delivers the interpolated, contributing values, which are continuously written back to memory. An alternative to directly write back the contributing lines to memory is to accumulate the lines in a buffer for the resulting segment. This accumulator is then written back to memory when all contributing segments have been processed. This reduces the memory traffic further.

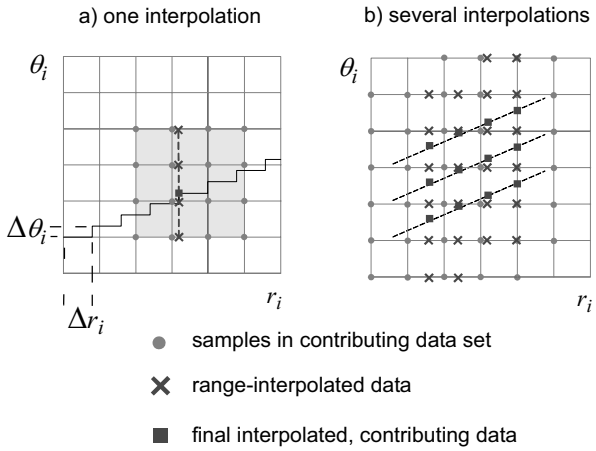


Figure 15. Using a two-dimensional cubic interpolation kernel to calculate one contributing data point.

The execution is described more in detail below. The different $<\Delta r_i,\Delta\theta_i>$ pairs are calculated based on the $<r_i,\theta_i>$ values at the region borders, and are assumed to be present in the steps below.

Range interpolation. The range-interpolated values are calculated by feeding the samples, read in a burst from the memory, through a one-dimensional cubic (i.e. length 4) interpolation kernel. The kernel delivers values which are collected in a vector, the length of which equals the number of points in a resulting segment data line. Table look-ups can be used here for high computational efficiency in the interpolation kernel. The interpolation kernel is further described below.

Tilting compensation. Before the beam-interpolations can take place, possible line tiltings must be compensated for. This is done by using a delay buffer with a triangular shape. The side proportions of the triangle are determined by the ratio between Δr_i and $\Delta\theta_i$. This delay buffer is a cost for having continuous execution and memory r/w on consecutive memory positions.

Beam interpolation. The elements of the resulting vector are calculated in parallel by feeding, via the delay buffer, the vectors created in the first step into an array of interpolation kernels. The resulting points from the interpolation array are placed in a vector which is written back to memory in a burst, or accumulated in a segment accumulator.

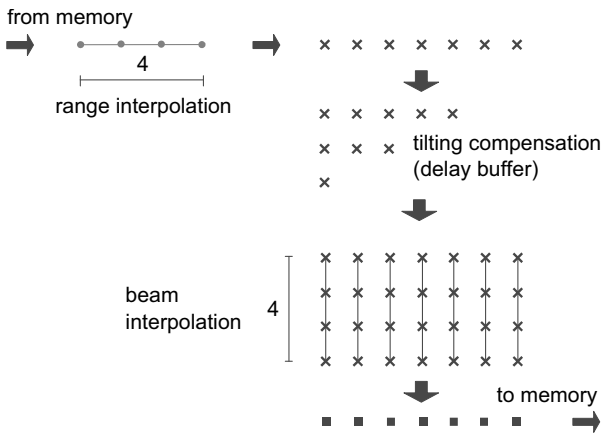
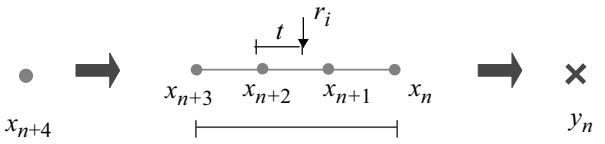


Figure 16. Execution scheme for interpolations.

The one-dimensional range interpolation kernel calculates a value y_n as a weighted sum of the values x_n, x_{n+1}, x_{n+2}, and x_{n+3}, as shown in Figure 17. The weights are acquired by table look-ups, exploiting the fact that the samples are equidistant. Four complex multiplications and three complex additions are required for each y_n. In addition, for each interpolation, the value r_i is calculated by adding Δr_i and a new t is calculated on the basis of the current r_i (since the read lines are parallel, the r_i and t values can, in the range interpolation, be reused from the previously processed line in the segment, with the cost of some extra buffer space). The beam interpolation kernel works analogously with the range interpolation kernel, but with the extension that the output point shall be adjusted (converted down to baseband – this adjustment is included to have a fair operation count comparison with other interpolation methods) by a complex multiplication.



$$y_n = x_n w_0(t) + x_{n+1} w_1(t) + x_{n+2} w_2(t) + x_{n+3} w_3(t)$$

weights are obtained by table look-ups

Figure 17. One-dimensional cubic interpolation.

Next we look into different ways of parallelizing these calculations and we derive figures for operation counts and memory accesses.

5.4 Parallelization of the calculations

The design freedom for parallelization on the fine-grain level is vast. An obvious choice is to have one processor per segment (as defined in Section 5.1) or set of segments. This parallelization level is close to the coarse-grain level. Within one segment, there is potential for having one range interpolation kernel and N beam interpolation kernels operating in parallel, see Figure 16. However, in this case the range interpolation kernel has to work N times faster than the beam interpolation kernels. Another choice is to have only one beam interpolation kernel, which serves all columns in a multiplexed way. It would then be as fast as the range interpolation kernel. The detailed implementation of the interpolation kernels should exploit the great inherent data parallelism by executing in a systolic or SIMD fashion.

5.5 Sample figures

The operation and read/write counts, together with buffer sizes, for creating *one* subaperture's contribution to a new 64-element data line are shown in Table 1. Cubic interpolation and nearest neighbor interpolation, respectively, are illustrated. The figures are based on the execution scheme described in Section 5.3 and include index generations and interpolations in the contributing data set, as well as transformations down to baseband. All interpolations and their associated calculations are assumed to be carried out directly against a local buffer. Data lines are transported from/to main memory only in bursts. A complex floating-point format with 8 bytes per data point is assumed. The size of the delay buffer is, in the extreme case when Δr_i and $\Delta \theta_i$ are equal, about $S^2/2$ points (=2048 when S=64). If a segment accumulator is used, the number of operations per memory access will increase by approximately 50% due to fewer memory accesses per resulting data point (3 reads +1 write, instead of 3 reads + 3 writes).

In the cubic interpolation, the interpolation weights are obtained with table look-ups. To create a line of 64 contributing data elements, 4608 operations, or 36 operations per memory access, are required. A weight table (1024 entries, four weights per entry) is of size 32 kilobyte. In the nearest neighbor interpolation case, 644 operations, or 5 operations per memory access, are required. Note, however, that investigations have shown that, in this case, upsampling is required in order to compensate for the simpler interpolation method. Upsampling with a factor of two in both directions to mitigate the effects of the simpler interpolation method, leads to a quadrupled total data size.

The operation count ratio between the cubic and the nearest neighbor interpolation approaches is 36/5, i.e. roughly eight. However, taking into consideration the 2x2 upsampling in the nearest neighbor case, the total operation count ratio is about two. Thus, a two times

reduction of the performance demands can be obtained by using the simpler interpolation method, but with the price of a four times larger memory. Consequently, there exists a memory/performance trade-off as illustrated in Figure 11. It can, however, be noted that, when having efficiently implemented sliding interpolation kernels, the performance cost for advanced interpolation must not always be as significant as indicated in the figure. There is a large potential for calculation reuse.

Table 1. The operation and read/write counts, together with buffer sizes, for creating one subaperture's contribution to a new data line.

linearization interval length = 64

	ops	mem. reads	mem. writes	data buffer (kB)	weight table (kB)	delay buffer (kB)
range interpol.	2112	64		0,5	32	
beam interpol.	2496		64	2,5	32	
total	4608			3	64	16
ops / mem. access	36			*cubic interpolation*		

	ops	mem. reads	mem. writes	data buffer (kB)	weight table (kB)	delay buffer (kB)
interpolation	644	64	64	0,5	0	16
ops / mem. access	5			*nearest neighbor interpolation*		

5.6 System balance

The achieved level of balance between computations and data transports in the system is an important factor. In a well balanced system a processor has the capacity to process data at a pace that corresponds to the rest of the system's capacity to exchange data with the processor. For a given system specification, this can be achieved by varying the different parameters in the parallelization design space. It is, of course, an advantage if only a few parameters need to be changed when specifications change. For example, it would be favourable if the signal processing system could be scaled by only changing the number of processing nodes on the coarse level, while the nodes themselves remain unchanged.

The radar data set and the created image are assumed to both have the size D $(=MN$ in Figure 3 times the size of a data point). The iterative execution of the FFBP, as illustrated in Figure 4, can be carried out in a ping-pong fashion, where the iteration results bounce between two memory banks, each of size D. The number of iterations is denoted I. The sustained data rate B is defined as D/T, where T is the integration time. As can be seen in Figure 18 the total memory bandwidth is $2B+(n+1)BI$, where the second term stems from the data transports during the iterations. n is the number of subapertures in a combination, which means that, for each created data point, n contributing points must be read. The required total computational performance is denoted by P. When making a partition on the coarse grain level, as described in to Section 3.1, each processing node will in principle have the above memory size, bandwidth, and performance figures, but divided by the chosen number of processing nodes, k. However, there will of course be effects from, e.g., the necessary data overlapping.

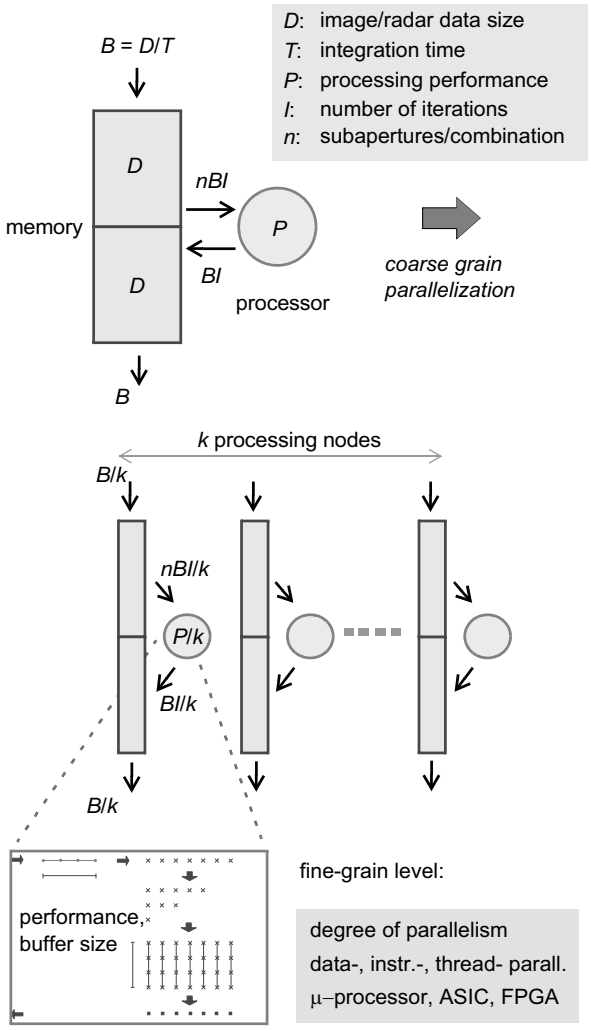


Figure 18. Conceptual view of architecture and design parameters. By a proper parallelization, system balance can be obtained. The fine-grain level shows a multitude of solution possibilities.

The fine-grain level shows a multitude of possible solutions. Various degrees and kinds of parallelism can be exploited, depending on, e.g., the system specification and the chosen implementation technology, such as microprocessor, ASIC, and FPGA.

Some example parameter values for the cubic interpolation case can be obtained from Table 1: If an operation is assumed to require 3 read/writes (2 reads+1 write) against a local register file, then the ratio between the register file bandwidth and the working memory (e.g. SDRAM) bandwidth is 108:1 (36x3:1). This points to the computational bandwidth, as well as the computational performance, that the fine-grain level architecture must support, depending on the speed of the working memory. As a comparison, the Stanford Imagine [8] shows (for an MPEG-2 encoder) a ratio of 470:1 between the local register files bandwidth and the SDRAM bandwidth. This ratio is in parity with the figures above and indicates that an architecture like Imagine may be proper on the fine-grain level.

5.7 Summary

Approaches for efficient fine-grain parallelization of the FFBP calculations have been studied. It is shown that it is possible to obtain an efficient memory operation, despite the complicated memory access patterns in the calculations. It is also shown that is possible to reduce the performance cost in the inherent memory size/performance trade-off in conjunction with the choice of interpolation method. This is done by exploiting the fact that the kernels are sliding and thus give possibilities for reuse of calculations, especially if the interpolation points in the contributing data set are aligned. Moreover, estimations have been made in order to get a view of the relative demands of the calculations in terms of processing performance and memory bandwidths.

6. Conclusions

New applications that put new demands on computer architectures show up all the time. Initially it is not known whether it is at all possible to meet the demands with state-of-the-art technology or foreseeable new technology. The SAR image forming described in this paper is an application of this kind. The algorithm has been fully implemented and tested in software, but needs parallel execution hardware to be useful in real time. Understanding the computational flow, with its associated memory, bandwidth and processing demands, is crucial in order to find the right kind of computer architecture for the class of problems. This paper illustrates the methodology and can be seen as an illustrative "up-stream" case study. We have analyzed the application in order to, in the first place, understand the algorithms, their different variations, and the challenges they present on a fundamental level. From this, we have found a basic approach (coarse-grain/fine-grain parallelization), and then further studied the demands and the design options of the interesting calculation kernels on the fine-grain level. Without pointing out any specific implementations, we have provided approximate models on which the demands are quantified.

7. References

[1] W.G. Carrara, R.S. Goodman, and R.M. Majewski, *Spotlight Synthetic Aperture Radar, Signal Processing Algorithms*, Boston: Artech House, 1995.

[2] F. Natterer, *The Mathematics of Computerised Tomography*, New York: Wiley, 1986.

[3] L.M.H. Ulander, H. Hellsten, and G. Stenström, “Synthetic-aperture radar processing using fast factorized back-projection,” *IEEE Transactions on Aerospace and Electronic Systems*, vol. 39, no. 3, 2003, pp.760-776.

[4] A. Olofsson, “Real-time signal processing in airborne ultra-wideband low-frequency SAR,” Master Thesis, Chalmers University of Technology, Göteborg, 2003. (In Swedish)

[5] M. Taveniku, A. Åhlander, M. Jonsson, and B. Svensson, "The VEGA moderately parallel MIMD, moderately parallel SIMD, architecture for high performance array signal processing," *Proc. 12th International Parallel Processing Symposium, IPPS´98*, Orlando, FL, USA, 1998, pp. 226-232.

[6] A. Åhlander and A. Åström, “A high speed signal processing system,” *Proc. Sixth Annual High Performance Embedded Computing Workshop, HPEC 2002*, MIT Lincoln Laboratory, Lexington, MA, USA, Sept. 24-26, 2002.

[7] P.-O. Frölind and L.M.H. Ulander, "Evaluation of angular interpolation kernels in fast back-projection SAR processing," *IEE Proceedings on Radar, Sonar and Navigation*, June 2006, Volume 153, Issue 3, pp. 243-249.

[8] U. Kapasi, S. Rixner, W. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. Owens, “Programmable Stream Processors,” *IEEE Computer*, August 2003, pp. 54-62.