Multiobjective Optimization of SAR Reconstruction on Hybrid Multicore Systems

Adeesha Wijayasiri, Tania Banerjee, Sanjay Ranka, Sartaj Sahni, and Mark Schmalz

***Abstract*—Hybrid multicore processors (HMPs) are poised to dominate the landscape of the next generation of computing on the desktop as well as on exascale systems. HMPs consist of general purpose CPU cores along with specialized coprocessors and can provide high performance for a wide spectrum of applications at significantly lower energy requirements per floating-point operations per second (FLOP). In this article, we develop parallel algorithms and software for constructing multiresolution synthetic aperture radar images on HMPs. We develop several load balancing algorithms for optimizing time performance and energy on HMPs. We also present a systematic approach for deriving the energy-time performance tradeoffs on HMPs in the presence of dynamic voltage frequency scaling. Pareto-optimal curves are presented on a system consisting of 24 traditional cores and a GPU.**

***Index Terms*—Dynamic voltage frequency scaling (DVFS), GPU, hybrid multicore processors (HMP), load balancing, list assignment, multiresolution images, power and energy evaluation, synthetic aperture radar.**

I. Introduction

SYNTHETIC aperture radar (SAR) image formation utilizes tensor product-based transformation of radar return pulse histories to yield a spatial representation containing possible target objects. Algorithms such as backprojection have been employed in reconstructing images from SAR pulse data to produce better quality reconstructions than frequency domain algorithms [1]–[3] due to support for higher resolution and fewer assumptions about the image, albeit with high computation time [4], [5]. Mulitresolution approaches for improving the time performance of sequential SAR algorithms were proposed in [6]. This is beneficial, for example, in change detection applied to reconstructed SAR video, where reduced resolution (and lower computational time) may be appropriate for background regions, while candidate target regions are rendered at higher resolution. Such a dynamic data-driven approach has the benefits of adaptively optimizing the overall computation requirements based on the nature of the underlying terrain.

Manuscript received February 14, 2020; revised May 18, 2020 and June 21, 2020; accepted July 9, 2020. Date of publication August 5, 2020; date of current version August 26, 2020. This work was supported by the Air Force Office of Scientific Research, under Contract FA9550-15-1-0047. This article was presented at the IEEE 24th International Conference on High Performance Computing (HiPC), Jaipur, India, Dec. 2017. *(Corresponding author: Adeesha Wijayasiri.)*

The authors are with the Department of Computer, and Information Science, and Engineering, University of Florida, Gainesville, FL 32611 USA (e-mail: adeeshaw@cise.ufl.edu; tmishra@cise.ufl.edu; ranka@cise.ufl.edu; sahni@cise.ufl.edu; mssz@cise.ufl.edu).

Digital Object Identifier 10.1109/JSTARS.2020.3014531

Hybrid multicore processors (HMPs) are poised to dominate the landscape of the next generation of computing on the desktop as well as in exascale systems [7]. HMPs consist of general purpose GPU cores along with specialized cores and are expected to provide benefits to a wide spectrum of applications at significantly lower energy requirements per floating-point operations per second (FLOP) [8], [9]. In this article, we describe a comprehensive strategy for efficiently implementing multiresolution SAR construction on HMPs consisting of a CPU and a GPU. We provide load balancing strategies for a combination of CPU and GPU cores. We show that, depending upon whether time performance or energy optimization is critical, separate load balancing strategies should be used on HMPs. We provide time performance, power, and energy evaluation as well as modeling and validation for a variety of CPU–GPU core combinations. We also discuss the multiple objectives of minimizing the runtime and energy required for SAR image reconstruction. Throughout this article we will use single precision computations when we present performance results.

The main contributions of this article are:

1) Efficient implementations of multiresolution SAR image reconstruction on a GPU and CPU. Our GPU implementation achieves 725 GFLOPS on an Nvidia Tesla K40 m GPU and the CPU implementation achieves 170 GFLOPS on a 24 core 2-socket Intel Xeon CPU E5-2695 V2 architecture.
2) The work distribution algorithm, developed by us in [6], is generalized to distribute workload among multiple CPU cores and the GPU. Due to varying spatial resolution and variable computational capabilities of CPU cores and GPU, the naive method of assigning an equal number of image partitions to each core do not necessarily yield optimal time.
3) Empirical models of runtime, power, and energy consumption of SAR image reconstruction on both CPU and GPU facilitate development of load balancing techniques for distributing work on an HMP system. These derived techniques can be applied to obtain either time performance or an energy optimal implementation.
4) A study of the impact of dynamic voltage frequency scaling (DVFS) on time performance-energy tradeoffs, under the assumption that the frequency of the CPU and GPU can be independently controlled.
5) A dynamic programming approach to obtain runtime, power, and energy consumption of SAR image reconstruction while reducing the exhaustive search space.

Algorithm 1: Algorithm for Sequential Backprojection.

Input: Pulse Array, Location Arrays, N_{fft}, W_r
Output: $N_x \times N_y$ image
1: **for** $i = 1$ to Number of Pulses in Pulse array **do**
2: Calculate IFFT for the pulse
3: Perform FFTShift for the result of IFFT and store as pData array.
4: **for** $j = 1$ to $N_x \times N_y$ **do**
5: Calculate ΔR w.r.t. pixel location.
6: let $index = \Delta R \cdot \frac{N_{fft}}{W_r}$ //Find the range bin that is closest to ΔR.
7: let $v1 = pData[index]$ *and* $v2 = pData[index + 1]$
8: Interpolate $v1$ and $v2$ to get the value (v) that corresponds to ΔR.
9: Multiply with phase correction.
10: Accumulate the calculated value for the current pulse with that value calculated for previous pulses.
11: **end for**
12: **end for**
13: Normalize computed pixel values.
14: **return**

The remainder of the article is organized as follows. In Section II, we describe the standard single-core backprojection algorithm for SAR image reconstruction as well as our GPU algorithm of [6]. In Section III, we apply the workload distribution algorithm developed by us in [6] to arrive at a multicore algorithm for SAR reconstruction. In Section IV, we extend the list assignment scheduling algorithm of [6] to arrive at a multilevel workload scheduling algorithm suitable for an HMP for both single resolution and multiresolution SAR images. In Section V, we present time performance, power, and energy evaluations for our SAR reconstruction methods on a CPU, a GPU, and a CPU–GPU hybrid system. A comparison of different load balancing schemes in terms of time performance and energy is provided in Section V, where we analyze time performance, power, and energy analysis on CPU and GPU for SAR reconstruction when DVFS is used. Finally, Section VI concludes the article.

II. Background and Previous Work

A. Backprojection Algorithm

Backprojection is a time domain reconstruction algorithm that inputs radar pulses collected in the frequency domain over a specific time interval, then sums their effects to produce a reconstructed output image [10]–[12].

Equations which were developed in [13] are used as the core of the sequential backprojection algorithm presented in Algorithm 1.

As described in [13], inputs for the Algorithm 1 are as follows.

1) *Pulse Array:* Contains phase history data.
2) *Location Arrays:* Contain (x, y, z) position of the sensor at each pulse.
3) N_{fft}: The length of the IFFT.
4) W_r: Maximum alias free range extent given by $c/2 \cdot \Delta F$, where c is speed of the light and ΔF is the frequency step size.

The differential range (ΔR) and phase correction used in Algorithm 1 can be calculated as described in [13]. FFTShift is a function that is used to shift zero-frequency components to the center of the spectrum [13].

An analysis of algorithm 1 reveals that backprojection takes $O(N_x \cdot N_y \cdot N_p)$ time to reconstruct an $N_x \times N_y$ image using N_p pulses. Algorithm 1 consists of two main for loops that can be used for parallelization: 1) A loop that iterates every pulse in the pulse array and 2) a loop that calculates the value of each pixel.

High resolution output images for realistic scene sizes can grow to several gigabytes in size. A key observation is that most of the scene information is not interesting for many practical applications. Objects are either not changing, or the nature of the change is already understood and expected. In areas meeting these constraints, the frequent generation of high resolution images is unnecessary. CPU time and power resources could be better allocated toward the production of imagery in other areas, where results are more likely to be mission-critical. High resolution renderings of the entire scene are only necessary at infrequent intervals, to detect small changes in areas that are being rendered in low resolution. An efficient, high-performance surveillance technology can exploit this observation.

For multiresolution SAR formation, one can consider the output image as decomposed into tiles where each tile has a potentially different resolution. The benefits of this approach follow from the fact that the computational complexity of backprojection scales proportionally with the output image size (in pixels). This means that identifying subsections of the image where activity is not occurring, and rendering those areas at low resolution, can significantly reduce the computational burden of generating high-performance surveillance imagery of a wide area. The decision about which pixels to render in high resolution can be made externally (for example, a change detection algorithm) but due to its complexity is outside the scope of this article. The key input is the target resolution value to be assigned to each tile of the output image based on the contents of this mask.

B. GPU SAR Reconstruction

SAR image reconstruction using backprojection demonstrates high parallelizability and GPUs can be used to speedup the SAR image reconstruction process [14]. Over the past decade, researchers have been investigating about efficient algorithms to reconstruct SAR images using parallel processing units. Chapman *et al.* introduced a parallel processing technique which distributes pulse array among thread blocks in the GPU [15]. Methods on implementing SAR backprojection with a single GPU has been explored in [16] where the partitioning of data along both range and aperture is allowed.

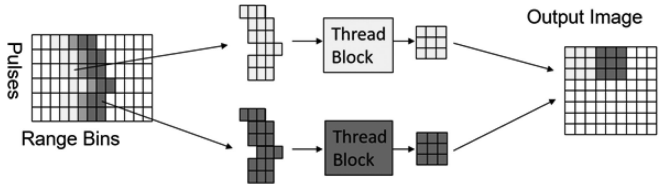


Fig. 1. Output image partitioning in a single GPU.

Algorithm 2: Kernel for Backprojection on a Single GPU.

Input: pData, N_{fft}, minF, Location Arrays
Output: $N_x \times N_y$ image
1: Calculate pixel index corresponding to blockIdx and threadIdx.
2: **for** $i = 1$ to Number of Pulses **do**
3: Calculate pixel value per line 5-9 of Algorithm 1
4: Accumulate the calculated value for this pulse with the value calculated for previous pulses.
5: **end for**
6: **return**

Nvidia's GPU architecture and programming model are described in [17]–[19]. Algorithm 2 gives the kernel code for backprojection on a single GPU and Fig. 1 shows how the pulses are distributed to each thread block in the GPU [6]. In this basic single GPU code, each thread computes a single pixel value, and each GPU core has to access all of the pulse data. The IFFT and FFTShift of the pulse data is computed and saved in the array $pData$ in GPU memory before the GPU kernel is invoked. CUDA's cuFFT library provides an efficient implementation of the IFFT function that can be used to calculate the IFFT in a separate kernel. The normalization of final pixel values is also done in a separate kernel.

Analysis using the Nvidia profiler reveals that the GPU kernel code requires 47 floating point operations (FLOPS) per pulse per pixel, with one-third of the time spent on actual computations and the remainder of memory access. According to Algorithm 2, each iteration requires six memory read accesses and one memory write access. Memory read accesses consist of four location array reads and two pulse array reads. The memory write access is for writing the accumulated value back to memory. In order to reduce the write accesses to device memory, we used a register variable to store accumulated values for each pulse; the final value of this register was then written to the device memory. This reduced device memory writes from $n^2 \cdot p$ to n^2.

III. Multicore SAR Reconstruction

A. CPU Optimizations

With the introduction of advanced vector extensions (AVX), CPUs are able to compute up to 8 32-bit registers concurrently [20], [21]. Contemporary compilers perform autovectorization on the CPU code to utilize the vector calculation capabilities on the CPU. In order to improve vectorization we explored different versions of the SAR image reconstruction

for i = 1 to N_x/T_x
  for j = 1 to N_y/T_y
    for k = 1 to T_x
      for l = 1 to T_y
        for p = 1 to N_p
          Calculate pixel location corresponds to i, j, k and l
          Calculate pixel values as stated in Algorithm 1

Fig. 2. Basic **for** loops in the SAR reconstruction algorithm. N_x and N_y refers to the image dimensions, T_x and T_y refers to the tile dimensions, and N_p is the number of pulses.

algorithm, for example, where the reconstructed image is partitioned into tiles as described in [6], and the for loops of Algorithm 1 are replaced by those of Fig. 2. The two outermost loops of this figure iterate through the image tiles in the x and y dimensions. The next two loops iterate through a tile's pixels in the x and y dimensions, and the innermost for loop iterates through the pulses. All possible combinations and/or fusions of these for loops were evaluated experimentally to find the combination/fusion that gives the best time performance. Our experiments show that fusing the outermost two loops into a single loop, fusing the next two loops into a single loop and placing the innermost loop as the second loop results in the best time performance. Algorithm 3 is the resulting backprojection algorithm.

In order to reduce memory transactions and improve cache efficiency, location data are read from the location array then stored in registers and used by the innermost for loop as in Algorithm 3. These optimizations led us to achieve 6 GFLOPS from a single core of an Intel Xeon CPU E5-2695 V2.

Algorithm 3 was coded in C and compiled using the Intel C compiler with -O3 optimization with autovectorization and fast math calculations and performance is increased to 7 GFLOPS from a single core of an Intel Xeon CPU E5-2695 V2.

B. Multicore Implementation

Algorithm 3 gives the pseudocode for backprojection on a multicore CPU system based on output image partitioning. The master process initiates the computation by allocating work to each core. The output image tiles produced by the slave processes are saved in main memory and the master process normalizes the computed pixel values to create the final output image.

Our experimental results indicate that the pulse partitioning approach is much slower than the image partitioning approach when Algorithm 3 is used. The main reason for this is that, in pulse array partitioning, each CPU core needs to access all of the output image array that resides in shared memory, using atomic transactions. Hence, we do not provide a muticore pseudocode.

IV. Hybrid SAR Reconstruction

Efficient distribution of work among the GPU and CPU cores is crucial to maximizing the utility of computational parallelism. We modified and used methods described in [5] for partitioning

Algorithm 3: Parallel Backprojection on a Multicore CPU System: Output Image Partitioning.

Input: pData, N_{fft}, W_r, Location Arrays
Output: $N_x \times N_y$ image
1: Master process allocates tiles to slave processes.
2: // Each process does the following
3: **for** $i = 1$ to Number of tiles allocated to the CPU core **do**
4: **for** $j = 1$ to Number of Pulses **do**
5: Read location data and store in register variables.
6: **for** $k = 1$ to Number of pixels in a tile **do**
7: Calculate pixel location corresponding to i, k and process ID.
8: Calculate pixel value as stated in line 5-9 of Algorithm 1
9: Update accumulated values for each pixel in register variables.
10: **end for**
11: **end for**
12: Write final pixel values to main memory
13: **end for**
14: Master process normalizes computed pixel values.
15: **return**

a uniform-resolution output image in terms of pulse partitioning and output image partitioning. Pulse partitioning involves distributing part of the pulse array to the GPU and the rest to the CPU, with subarray partitioning inside each CPU core if required. This reduces CPU to GPU communication time due to pulse data movement across different levels of the memory hierarchy. However, the entire reconstructed image must be transferred back to the CPU memory and merged with the image calculated by the CPU cores.

A. Partitioning for Uniform Resolution

Output image partitioning can be implemented as round robin (naive method), as well as static or dynamic range dimension partitioning. Round robin removes image tiles from a scheduling queue and assigns them to the GPU or the CPU core that has the lowest current workload, but requires the entire pulse array to be sent to the GPU. Static range dimension partitioning assigns output image partitions to the GPU or to the CPU core that is associated with similar distance of the reconstructed tile from the center of the reconstructed scene. This assignment method reduces the pulse array communication time and is refined by dynamic range dimension partitioning, which reallocates tiles when the look angle changes [5].

For SAR reconstruction on a hybrid system, we implemented four methods to distribute the workload between the GPU and CPU.

1) Tile-based partitioning for GPU/CPU level and tile-based partitioning among CPU cores.
2) Pulse-based partitioning for GPU/CPU level and tile-based partitioning among CPU cores.
3) Tile-based partitioning for GPU/CPU level and pulse-based partitioning among CPU cores.
4) Pulse-based partitioning for GPU/CPU level and pulse-based partitioning among CPU cores.

Algorithm 4: CPU–GPU level Tile-Based and CPU-Core Level Pulse-Based Partitioning on a Hybrid System.

Input: pData, N_{fft}, W_r, Location Arrays
Output: $N_x \times N_y$ image
1: Master process allocates tiles for the GPU and CPU cores.
2: GPU host process transfers data arrays to the GPU memory and executes Algorithm 2 to compute allocated tiles for the GPU.
3: In parallel to GPU, each CPU core runs Algorithm 3 on the tiles allocated for the CPU.
4: GPU host process collects the constructed image from GPU memory.
5: Master process normalizes computed pixel values.
6: **return**

Algorithm 5: CPU-GPU Level Pulse-Based and Cpu-Core Level Tile-Based Partitioning on a Hybrid System.

Input: pData, N_{fft}, W_r, Location Arrays
Output: $N_x \times N_y$ image
1: Master process divides pulses for the GPU and CPU cores.
2: GPU host process transfers data arrays to the GPU memory and executes Algorithm 2 to compute the full image using pulses allocated for the GPU.
3: In parallel to GPU, each CPU core runs Algorithm 3 on the pulses allocated for the CPU to compute the full image.
4: GPU host process collects the constructed image from GPU memory.
5: Master process finalizes the image by summing up the two images made by the GPU and CPU cores and normalizing pixel values.
6: **return**

Algorithm 4 presents the basic pseudocode for tile-based partitioning for GPU/CPU level and tile-based partitioning among CPU cores. The master process initiates the process by allocating workload to the GPU and CPU cores and sending the pulse and location arrays to the GPU. The tiles computed by the GPU are sent back to the main memory. Finally, the pixels in all tiles are normalized.

Algorithm 5 shows the pseudocode for pulse-based partitioning for GPU/CPU level and tile-based partitioning among CPU cores. The master process distributes pulses to the GPU and CPU cores to reconstruct a separate image from those pulses. After computations are done, the images produced by the GPU and CPU cores will be added and normalized by the CPU to complete the final image.

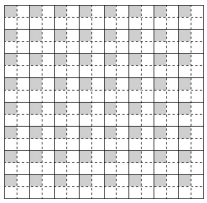


Fig. 3. 16×16 tile with 2×2 grid overlay where pixels to be computed are shaded.

Methods 3 and 4 for workload distribution require pulse-based partitioning among CPU cores. Due to the reasons given in Section III, these methods do not perform well and are not considered further in this article.

B. Partitioning for Multiresolution

Multiresolution SAR image reconstruction begins by partitioning the reconstructed image into equal-size tiles. Each tile has a designated required resolution. At highest resolution (i.e., a resolution of 1), every pixel in the tile is to be reconstructed from pulse data. In one-fourth resolution, we overlay a 2×2 grid over the tile and compute only the top left pixel in each 2×2 grid cell. Fig. 3 shows a 16×16 tile with a 2×2 grid overlay. The pixels to be computed are shaded. At one-sixteenth resolution, the overlay is done using a 4×4 grid and the top left pixel in each grid cell is to be computed.

Our main objective in this article is to determine work allocation for the GPU and CPU cores to reconstruct the image using a CPU–GPU hybrid system when multiple resolution levels are required in the output image.

Algorithms 4 and 5 have been optimized for SAR image reconstruction with multiple resolution levels. The objective of the work distribution is to balance the computational load proportionally across the GPU and all CPU cores. For the pulse distribution method described in Algorithm 5, the presence of multiple resolution levels in the image do not affect the work distribution between the GPU and CPU; it affects only the tile distribution among CPU cores. In Algorithm 4, a tile distribution strategy that distributes work load to the GPU proportional to the computing power of the GPU is required. Since multiple resolution levels are considered, the tile distribution strategy should not only consider the number of tiles but also the work inside the tiles. We used the list assignment algorithm (LA) described in [6] to distribute work between GPU and CPU cores. As proven in [6], when work is distributed using LA, the imbalance is at most that corresponding to one tile, which is negligible when the number of tiles is large.

V. Experimental Results

In Section V-A, we describe our experimental platform. In Section V-B, we compare the time performance as well as power and energy requirements using tile-based and pulse-based partitioning on CPU cores. In Section V-C, we give the time performance, power, and energy results for a GPU and in Section V-D we do this for a hybrid CPU–GPU system. In Section V-E we present time performance, power, and energy results on CPU cores and for a GPU when DVFS is used. All experimental results reported in this section exclude the normalization step of the SAR backprojection algorithm.

TABLE I
Execution Time (Seconds) Using 23 Cores. High, Medium, and Low Correspond to Full Resolution, $1/4$th, and $1/16$th of Full Resolution Respectively

Image size	Pulses	High		Medium		Low	
		tile	pulse	tile	pulse	tile	pulse
2048×2048	1000	1.37	2.20	0.34	0.39	0.086	0.10
	3000	4.25	6.7	1.07	1.18	0.28	0.30
	5000	7.15	11.38	1.83	2.05	0.5	0.52
4096×4096	1000	5.5	8.8	1.38	1.54	0.34	0.40
	3000	16.79	26.6	4.22	4.63	1.06	1.18
	5000	28.27	44.6	7.11	7.81	1.84	2.05

A. Target Platform

Our hybrid test platform comprises an Intel Xeon E5-2695 V2 CPU and a Tesla K40 m GPU. The Tesla K40 m has 13 multiprocessors and each multiprocessor has 192 CUDA cores. The default core clock speed is 745 MHz with a maximum boosted clock speed of 875 MHz. The GPU card has 6 4-K registers per block and up to 48 KB of shared memory per block. The Nvidia System Management Interface (nvidia-smi) command line utility is used for power and energy measurements with CUDA SDK 8.0. The Intel Xeon CPU E5-2695 V2 is a 24-core system where each socket has 12 cores. It has 2.4-GHz base frequency with a 115-W thermal design point (TDP). It has 3.2 GHz of max turbo frequency and for Section V-B and Section V-D measurements are taken with turbo boost enabled with maximum frequency. For Section V-E, turbo boosting is disabled to measure performance for the exact CPU core frequency. The Intel Xeon CPU E5-2695 V2 has 30720 KB of cache with 64 byte alignment and a memory of 768 GB. PowerGadget, which is based on the running average power limit (RAPL) utility, is used to measure power.

B. CPU Results

Table I presents the execution times for tile-based and pulse-based partitioning using only the CPU of our Xeon platform. As can be seen, tile-based partitioning results in better performance on our multicore CPU. In fact, tile-based partitioning was up to 36% faster than pulse-based partitioning on our datasets.

Figs. 4–7 give the runtime, power, and energy for SAR reconstruction using tile partitioning as a function of number of cores, number of pulses, image size, and resolution. The SAR reconstruction was done using only the CPU of our Xeon platform and the stated parameters were selected as below. The tile size was set to 16×16.

1) *Number of Cores:* We used 1, 2, 4, 12, and 23 CPU-cores for the experiments. We did not use all 24 cores in this experiment, to better compare with our hybrid results where one core was dedicated to GPU management.
2) *Number of Pulses:* The number of pulses used in our experiments were 2000, 3000, and 4000.

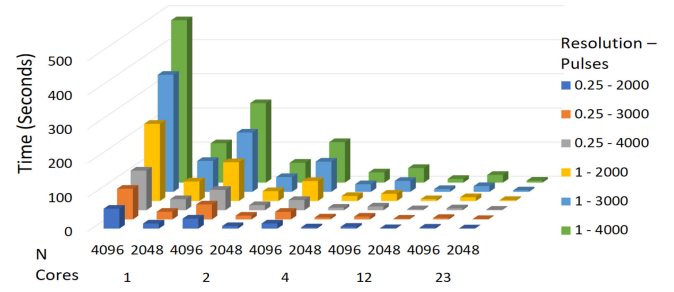


Fig. 4. CPU time for varying image sizes, number of pulses, and resolutions.

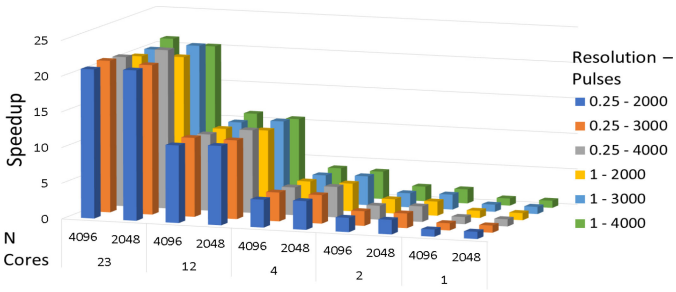


Fig. 5. Speedup for different number of cores with varying images sizes, number of pulses, and resolutions.

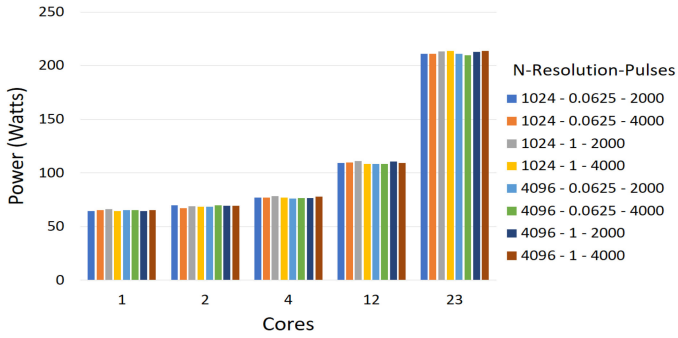


Fig. 6. CPU power when different number of cores are used.

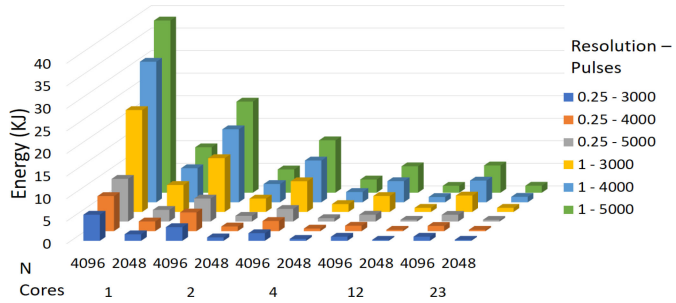


Fig. 7. CPU energy consumption for different number of cores. Energy is measured with varying image sizes, number of pulses, and resolutions.

3) *Image Size :* Two sizes for the reconstructed SAR image were used–2048×2048 and 4096×4096 pixels.
4) *Resolution Tuple:* This tuple represents the percentage of each resolution level that we used in multiresolution SAR image reconstruction. We used a 3-tuple, (a, b, c) where a, b, and c denotes high, medium, and low resolution percentages corresponding to full resolution, $1/4$th, and $1/16$th of full resolution, respectively. In the experiments of Figs. 4–7, we used the tuples (100,0,0), (0,100,0), and (0,0,100).

Fig. 4 shows CPU processing time for SAR reconstruction when uniform resolution level is used with varying input parameters. Our experiment indicates that runtime is linearly proportional to the resolution level as well as to the number of pixels. This is not surprising, as with the resolution tuples used the workload is $O(n^2 r N_p)$, where n^2 is the number of pixels, r is the resolution corresponding to the only nonzero entry in (a, b, c), and N_p is the number of pulses. Although one may also expect a linear dependence on the number of pulses, runtime increased at a slightly faster rate than linear. This is due to higher cache miss rate resulting from an increase in pulses.

As the number of cores is increased, workload per core decreases. Fig. 5 gives the speedup obtained; for 23 cores, the speedup is more than 20.

Power consumption for memory and processor are major causes of CPU power consumption. Our experimental results revealed that DRAM power consumption has a constant value of 22.5 Watts per socket. We will use the term CPU power to refer the total of DRAM power and processor power hereafter. Fig. 6 shows that the CPU power variation depends mainly on the number of cores used. Starting with a baseline power consumption of 60 W, power increases linearly.

Fig. 7 shows that even though the power required increases with the number of cores, the energy consumed decreases in most cases. This is due to the corresponding decrease in runtime. Although power increases linearly starting from a base of 60 W and runtime decreases slightly less than linearly, energy consumption usually decreases because of the high 60-W base.

C. GPU Results

Our GPU implementation used tile-based partitioning and the parameters number of pulses, image size, and resolution were chosen as shown below. The GPU experiments reported in this section did not use the CPU cores to compute any tile.

1) *Number of Pulses:* Our experiments used 5000 and 10 000 pulses.
2) *Image Size:* Image size was 4096×4096, 8192×8192, and 16384×16384 pixels.
3) *Resolution Level:* Resolution levels were the same as for our experiments with the CPU (Section V-B).

Figs. 8 and 9, respectively, give the time, power, and energy measurements from our experiments.

As shown in Fig. 8, reconstruction time increased linearly with the number of pulses as well as with the number of pixels in the reconstructed image. Further, runtime is linear as a function of resolution (i.e., when $(a, b, c) = (0,100,0)$, runtime is (approximately) 1/4th that when $(a, b, c) = (100,0,0)$ and (approximately) 4 times that when it is (0,0,100).

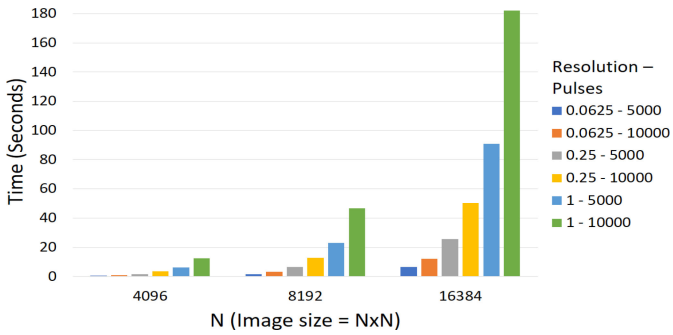


Fig. 8. Time Performance of SAR reconstruction on GPU.

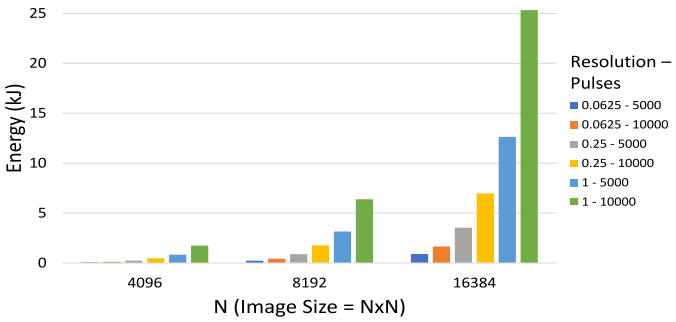


Fig. 9. Energy consumption on a GPU while varying image size, number of pulses, and resolution level.

The average power, approximately 165 W, required by the GPU for the duration of the program run, was relatively insensitive to image size and resolution; there is a very slight increase with the number of pulses.

Fig. 9 shows the energy consumed by our GPU. As expected, energy increases with increasing number of pulses, image size, and resolution level. Since GPU power is nearly constant, GPU energy is (approximately) linearly proportional to GPU time. Our GPU implementation achieved 725 Gflops on the platform described in Section V-A. This platform has 4.29 Tflops theoretical peak performance and CUBLAS matrix multiplication achieves 1470 Gflops on our platform.

D. Hybrid CPU-GPU Model

For our hybrid CPU–GPU experiments, we developed a model to predict time performance, which used linear regression on the data of Section V-B and Section V-C to obtain (1) and (2) to estimate runtime on our CPU and GPU, respectively. Here, N is the image size, N_p is number of pulses, N_c is the number of cores, and runtimes are for tile-based partitioning. Microsoft Excel was used to run the regression with 0.999 adjusted R^2 observed for each equation. Note that the number of pixels being computed is N^2 (i.e., full resolution)

$$T_{CPU} = 0.275 + 6.951 \times 10^{-9} \times N^2 \times N_p \times {N_c}^{-1} \quad (1)$$

$$T_{GPU} = 0.96 + 6.841 \times 10^{-11} \times N^2 \times N_p. \quad (2)$$

From the above equations, we derive (3) and (4) that estimate runtime for multiresolution images. Again, the estimated time is for tile-based partitioning

$$T_{nCPU} = T_{CPU} \cdot (a + b \times 0.25 + c \times 0.0625)/100 \quad (3)$$

$$T_{nGPU} = T_{GPU} \cdot (a + b \times 0.25 + c \times 0.0625)/100 \quad (4)$$

where (a, b, c) denotes the resolution tuple. Recall that a, b, and c are, respectively, the percent of tiles being computed at full, 1/4th and 1/16th resolution.

The runtime T_{hybrid} time for a hybrid computation (i.e., one that uses both the GPU and the CPU cores) is approximated by

$$T_{hybrid} \approx \max\{T_{nCPU}, T_{nGPU}\}. \quad (5)$$

To minimize the overall runtime on a hybrid system, we partition the workload between the CPU and GPU, so that $T_{nCPU} \approx T_{nGPU}$. Although (1)–(4) are for tile-based partitioning, when attempting to equalize T_{nCPU} and T_{nGPU}, we have the option of doing either pulse- or tile-based partitioning between the CPU and GPU. So, for example, if we use the CPU to compute all tiles using N_{CPU} pulses, then the GPU does this for the remaining $N_p - N_{CPU}$ pulses. In this case, N_p in (1) and (2) is replaced by N_{CPU} and $N_p - N_{CPU}$, respectively. Tile-based partitioning is used to divide the workload among the CPU and GPU cores.

We evaluated the accuracy of using the prediction model defined by (1)–(5) to obtain optimal workload partitions. For this evaluation, the true optimal workload partition was determined by exhaustively trying different workload partitions in the neighborhood of the partition obtained from the model. Note that the model only gives the fraction of the total workload that should be allocated to the CPU and GPU, which can be achieved by either using pulse partitioning or tile partitioning. Table II gives the workload partitioning and predicted runtime using the model for the case when the image is to be computed at full resolution [i.e., $(a, b, c) = (100,0,0)$] as well as the workload partitioning and corresponding measured runtime for an optimal partitioning. For the latter runtime, the CPU and GPU computed the pixels in their assigned tiles for using all pulses. So, for a 4096×4096 image using 23 cores and 40 000 pulses, our model predicted that best time performance will be obtained by having the CPU do 18% of the work, yielding a runtime of 38 s. The exhaustive search in the neighborhood of an 18% allocation to the CPU resulted in optimal time performance when the CPU was assigned only 16% of the work; the exhaustive search distributed work between the CPU and GPU using tile partitioning. The column labeled

TABLE II
PREDICTED AND MEASURED OPTIMAL WORKLOAD PARTITIONING AND RUNTIME FOR FULL RESOLUTION RECONSTRUCTION

cores	Image size (pixels)	Model		Optimum		Deviation of time(%)
		$cpu\%$	$time(s)$	$cpu\%$	$time(s)$	
1	4096×4096	1	46	0.7	44.2	3.91
	8192×8192	1	182	0.7	174.5	4.12
	16384×16384	1	728	0.8	689	5.36
23	4096×4096	18	38	16	37.72	0.74
	8192×8192	18	150	15	148.9	0.73
	16384×16384	18	599	15	599.1	0.65

TABLE III
PREDICTED AND MEASURED OPTIMAL WORKLOAD PARTITIONING AND RUNTIME(S) FOR MULTIRESOLUTION RECONSTRUCTION

Exp	Image size	Model		Tile based		Pulse based		D_t	D_p
		cpu%	*time*	*cpu%*	*time*	*cpu%*	*time*		
1	8192	19.29	13.24	17.00	12.96	16.79	12.8	3.29	3.29
	16384	18.67	50.44	16.80	51.06	16.79	50.69	0.50	0.50
2	8192	20.26	6.57	17.50	6.17	17.60	6.17	6.04	6.04
	16384	18.91	23.76	17.00	24.11	17.50	24.4	2.67	2.67

TABLE IV
OPTIMAL HYBRID FULL RESOLUTION RECONSTRUCTION OF A 16384×16384 IMAGE USING 40 000 PULSES

Cores	% load on CPU	Time(s)	Energy (KJ)
1	0.8	688.46	139.583
2	1.55	685.62	140.764
4	2.87	675.9	143.081
12	8.08	643.1	158.194
23	14.5	594.9	212.780

"Deviation of time" gives the percent deviation in the measured optimal runtime from that predicted by the model. For these tests, measured deviation from the model is within 6% when only 1 core is used and within 0.75% when 23 cores are used.

Table III gives the model predictions and measured runtimes for multiresolution reconstruction using 23 cores. In experiment 1 (Exp 1), $(a, b, c) = (60,20,20)$ and in experiment 2, it is (20,30,50). Both experiments used 40 000 pulses. For a 8192×8192 pixel image with $(a, b, c) = (60,20,20)$ (Exp 1), our model predicts optimal time performance when the CPU is assigned 19.29% of the work; the predicted runtime is 13.24 s. An exhaustive search in the neighborhood of 19.29% work being assigned to the CPU (the CPU is assigned x% of the tiles, where $x \simeq 19.29$) resulted in optimal time performance when the CPU was assigned 17% of the tiles; the optimal runtime was 12.96 s. When the neighborhood search was done by assigning x% of the pulses ($x \simeq 19.29$), best time performance was observed when $x = 16.79$%; the runtime for this x was 12.8 s. D_t and D_p give the deviation in the measured optimal times using the tile and pulse-based exhaustive searches relative to the model predicted optimal runtime, respectively. The maximum deviation in our experiments was 6.04%, indicating that our model is accurate within practicable constraints.

Table IV gives the measurements for an optimal hybrid full resolution reconstruction of a 16384×16384 pixel image using 40 000 pulses. For example, when the number of cores is 23, 14.5% of the work is done on the CPU and the balance (85.5%) on the GPU; this reconstruction took 594.9 s and consumed 212 780 joules of energy. The time-energy data of Table IV is plotted in Fig. 10 to obtain the pareto-optimal front for time and energy tradeoff. The percentage of the load processed by the CPU cores and the GPU for each of the data points in Fig. 10 is given in Table IV. These work distributions are observed when the optimum runtime is achieved.

We present a pareto-optimal front for energy and time performance trade-off in Fig. 10. Every point on this curve can be reached by changing the loading on the CPU and GPU, and this can be used to achieve the required energy savings by trading time performance.

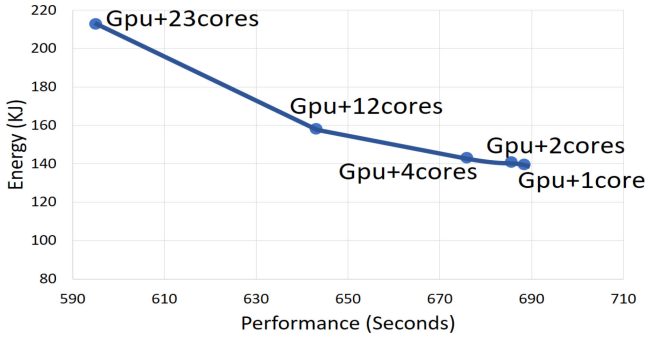


Fig. 10. Time performance and energy tradeoffs in our hybrid system. Both CPU and GPU are set to run at the highest frequency.

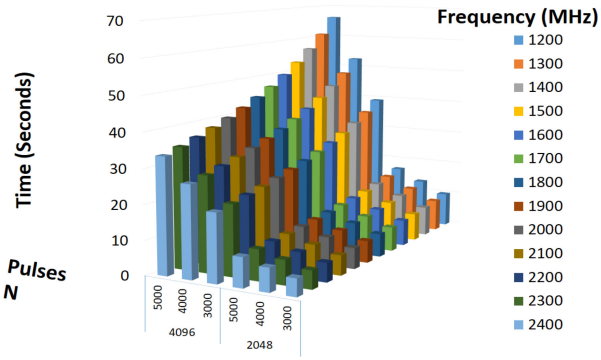


Fig. 11. Processing time using 23 cores and highest resolution for different image sizes and number of pulses.

E. Dynamic Voltage and Frequency Scaling

DVFS is used to adjust voltage and frequency dynamically to reduce power usage of a processor. Dynamic power and leakage power are the two main power consumption methods in a CMOS processor [22], [23]. Dynamic power is mainly utilized for execution of instructions and is linearly proportional to the frequency and to the square of the voltage [24], [25]. The idea of using DVFS was first introduced by Weiser *et al.* [26]. A critical analysis on energy management using DVFS is discussed in [27] where the author described categories of applications which can be benefited from DVFS.

We investigated the time performance, power, and energy behaviors with varying clock frequencies for both CPU and GPU, and in this section we present a DVFS model which can be used to obtain energy and time performance tradeoffs.

1) DVFS CPU Results: Figs. 11– 13 present SAR reconstruction runtime, power, and energy when tile-based partitioning was employed. 3000, 4000, and 5000 pulses were used with 2048×2048 pixel and 4096×4096 pixel image sizes. CPU clock frequencies used in our experiments were 1200 to 2400 MHz, with step size of 100 MHz. Number of cores were set to 23 and resolution was set to the highest level.

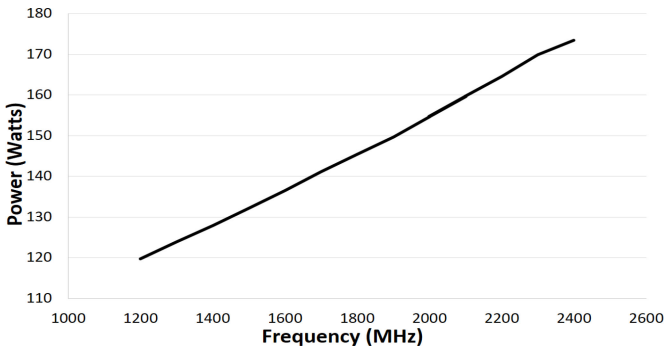


Fig. 12. Average CPU power using 23 cores with varying frequency levels.

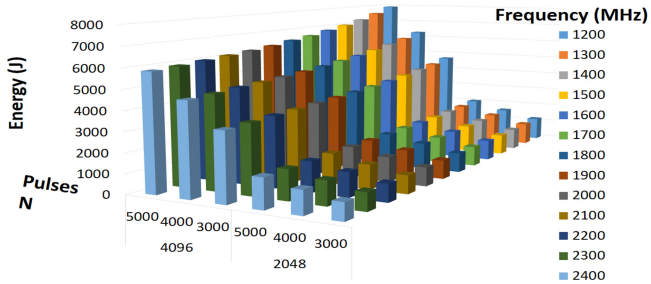


Fig. 13. CPU energy consumption with varying frequencies, image sizes, and number of pulses.

From Fig. 11, we see that runtime is inversely proportional to the CPU clock frequency. This is expected as the number of instructions executed per second is linearly proportional to the CPU clock frequency. Fig. 12 shows how the power varies with the frequency when 23 CPU cores are used. Fig. 13 gives the energy consumption for different frequencies for the experiments discussed above. Note that energy consumption decreases with frequency.

2) DVFS GPU Results: We measured time performance, power, and energy for our GPU implementation of our SAR reconstruction algorithm for varying GPU clock frequencies for various number of pulses and image sizes: $666, 745, 810$ and 875 MHz were used as frequency levels with 10 000 and 5 000 pulses, and $4096 \times 4096, 8192 \times 8192$, and 16384×16384 pixel image sizes.

Fig. 14 presents runtime as a function of GPU clock frequency, number of pulses and image size. As expected, runtime is inversely proportional to the GPU frequency. As shown in Fig. 15, GPU power increased linearly with frequency. This behavior is expected, as dynamic power is linearly proportional to the frequency. Energy consumed by the GPU is decreased sublinearly with the frequency decrement as shown in Fig. 16. Even though time increased with the frequency decrement, decreased in power resulted in overall decrease in energy.

3) Time Performance and Energy Tradeoffs for DVFS: Fig. 17 shows the energy-time performance tradeoff when different frequencies and cores are used for CPU and GPU. 4096×4096 pixel image size is used with 5000 pulses. When high frequencies are used, we observe a decrease in runtime

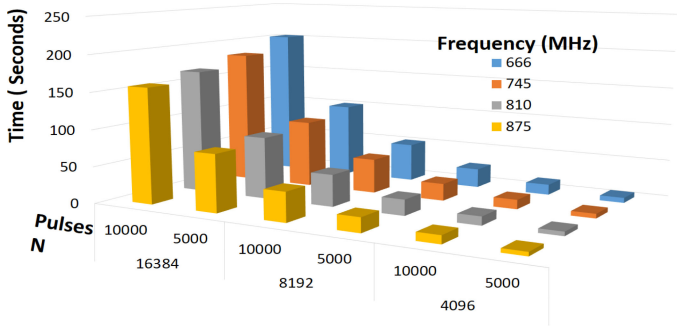


Fig. 14. GPU time for highest resolution with varying frequencies, image sizes, and number of pulses.

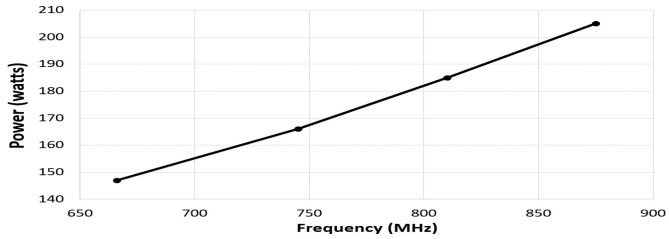


Fig. 15. GPU average power for different clock frequencies.

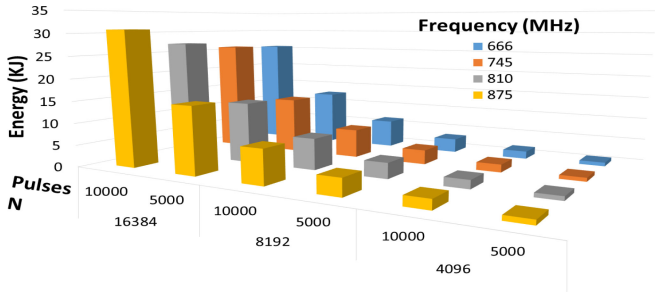


Fig. 16. Energy consumption on a GPU for various clock frequencies.

but an increase in energy consumption. This graph presents a pareto-optimal front for energy and time performance.

When the timing constraints are introduced, the energy-time performance tradeoff can be stated as an energy optimization problem

$$\begin{aligned} \underset{X_{CPU}, X_{GPU}}{\text{minimize}} \quad & E_{CPU}(N_c, f_c) + E_{GPU}(f_g) \\ \text{subject to} \quad & T_{CPU}(N_c, f_c) \leq T_1 \\ & T_{GPU}(f_g) \leq T_1 \\ & X_{CPU} + X_{GPU} = 1 \\ & N_c \in \{1, 2, 4, 12, 23\} \\ & f_c \in \{1200, 1300, 1400, \ldots, 2400\} \\ & f_g \in \{666, 745, 810, 875\} \end{aligned}$$

where N_c is the number of CPU cores, T_1 is the target runtime, X_{CPU} and X_{GPU} are the, respective, CPU and GPU workload, f_c is the CPU frequency and f_g is the GPU frequency.

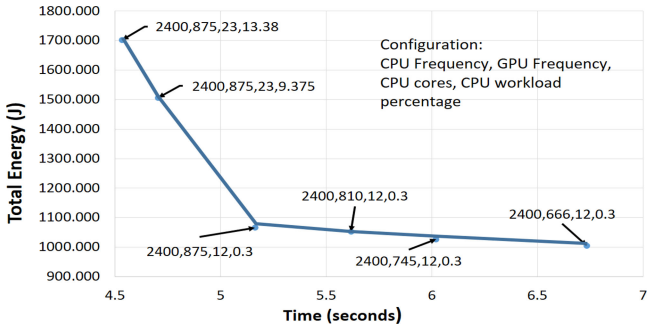


Fig. 17. Time performance and energy tradeoff in the hybrid system when different frequencies and cores are used.

$E_{CPU}(N_c, f_c)$, $T_{CPU}(N_c, f_c)$, and $E_{GPU}(f_g)$, $T_{GPU}(f_g)$ are the energy and time consumed by N_c CPU cores under f_c frequency and GPU device under f_g frequency, respectively. We developed a model to predict time and energy performance using regression analysis for the results shown in Section V-E, as follows:

$$T_{CPU} = a_1/(N_c \cdot f_c) \tag{6}$$

$$T_{GPU} = a_2/f_g \tag{7}$$

$$E_{CPU} = a_3 + a_4 \cdot \lceil N_c/12 \rceil/(N_c \cdot f_c) \tag{8}$$

$$E_{GPU} = a_5 - a_6/f_g \tag{9}$$

where $a_1 = 2.185 \times 10^{-5} \cdot P, a_2 = 5.418 \times 10^{-8} \cdot P, a_3 = 2.933 \times 10^{-8} \cdot P$, $a_4 = 0.001 \cdot P$, $a_5 = 1.497 \times 10^{-8} \cdot P$, $a_6 = 1.92 \times 10^{-6} \cdot P$. Here $P = N^2 \cdot N_p$ where N is the image size and N_p is number of pulses. Microsoft Excel was used to run the regression with above 0.99 adjusted R^2 for each of the equation.

It is possible to predict the minimum energy by solving the above optimization problem under the given time constraint, providing CPU and GPU workload percentages with CPU and GPU frequencies and the number of CPU cores when the minimum energy configuration is achieved.

We used this method to find energy-time performance tradeoff for a SAR image reconstruction of 4096×4096 pixels with 5 000 pulses. The MATLAB linprog function was used to solve the problem repeatedly for each value of N_c, f_c, and f_g, to find the minimum energy for all possible configurations. When we set $T_1 = 4.7$ s, theoretical results indicate that a minimum energy of 1504.2 J can be achieved with $N_c = 12$, $f_c = 2400$, and $f_g = 875$ by allocating 9.5% workload to CPU cores and 90.5% workload to the GPU. Experimentally, this configuration consumed 1486.8 J. Energy and the runtime for the best runtime configuration is 1702 J and 4.53 s, respectively. So it is possible to get 12.64% more energy efficiency by sacrificing only 3.75% runtime. Furthermore, if the time limit is set to 6.1 s, our theoretical model yields a configuration of $N_c = 12$, $f_c = 2400$, and $f_g = 745$, by allocating 0.3% workload to CPU cores and 99.7% workload to the GPU with a minimum energy consumption of 1052 J. Experimentally, energy consumed for this configuration is 1026 J yielding 39.72% energy efficiency by sacrificing 34.66% on runtime.

TABLE V
COMPARISON OF RUNTIME AND ENERGY BETWEEN OUR MODEL AND ACTUAL RESULTS

Configuration	Runtime(s)			Energy(J)		
	Est.	Act.	Err.(%)	Est.	Act.	Err.(%)
2400,875,23,13.38	4.5	4.53	0.66	1679	1702	1.35
2400,875,23,9.38	4.707	4.704	-0.06	1498	1507	0.6
2400,875,12,0.3	5.18	5.163	-0.32	1084	1066	-1.69
2400,810,12,0.3	5.59	5.62	0.53	1070	1053	-1.61
2400,745,12,0.3	6.08	6.02	-1.0	1052	1026	-2.53
2400,666,12,0.3	6.80	6.73	-1.04	1027	1004	-2.29

To validate the derived time and energy values from our model, we compared actual runtime and energy with estimated runtime and energy obtained using (6) to (9). A 4096×4096 pixel image was used with 5000 pulses. Table V shows the estimated and actual time and energy results and the error percentage for each case. Configuration column represents CPU frequency, GPU frequency, CPU cores, CPU workload, respectively. According to Table V, error percentage between estimated and actual runtime and energy is less than 3%.

F. Dynamic Programming Model

Dynamic programming model can be used to solve the resource allocation optimization problem efficiently [28]. An analysis on thermal versus energy optimization for DVFS-enabled processors is presented in [29] and a discussion on resource allocation via dynamic programming in activity networks is presented in [30]. In this section, we present an optimization strategy based on dynamic programming for the SAR image reconstruction on hybrid multicore system.

Consider a system that has N_c number of CPU cores, N_g number of GPU cores, F_c number of CPU frequency levels, and F_g number of GPU frequency levels. Energy minimization problem for this system can be written as

$$\begin{aligned} \text{minimize} \quad & E(n_c, f_c, n_g, f_g, S) \\ \text{subject to} \quad & T(n_c, f_c, n_g, f_g, S) \leq T \\ & n_c \in N_c, n_g \in N_g, f_c \in F_c, f_g \in F_g \end{aligned}$$

where S is the total problem size, $T(n_c, f_c, n_g, f_g, S)$ is the total execution time for the problem and T is a given time limit. If we assume discrete distribution of workload among CPUs and GPUs, we can denote the cardinality of the set of workload distribution s as X. As an example if the workload distribution set $W = \{0.000, 0.001, 0.002, \ldots 1.000\}$, then $X = 1000$. In order to determine the best energy optimization configuration for this system, number of evaluations required is

$$T = N_c \cdot N_g \cdot F_c \cdot F_g \cdot X. \tag{10}$$

When the parameters used in (10) become large enough, total number of evaluations can be practically infeasible. In this section we proposed a dynamic programming approach to make these evaluations more feasible by decomposing the problem into separate subproblems. The energy and time consumed by the CPU cores is independent of the energy and time consumed

by GPU cores and depends with the workload processed by each core, which made this decomposition possible.

The energy consumed by CPU cores (E_{CPU}) is a function of the number of CPU cores used (n_c), the CPU frequency (f_c), the CPU workload fraction (x_c), and can be written as $E_{CPU}(n_c, f_c, x_c)$. The energy consumed by GPUs (E_{GPU}) is a function of the number of GPUs used (n_g), the GPU frequency (f_g), the CPU workload fraction (x_g), and can be written as $E_{GPU}(n_g, f_g, x_g)$.

The time taken by the CPU cores to process a given computational workload, is given by T_{CPU} which is the maximum time taken by a CPU core to process $x_c \cdot S/n_c$ amount of load. Similarly, the time taken by the GPUs to process the load is given by T_{GPU} which is the maximum time taken by a GPU to process the remaining $x_g \cdot S/n_g$ amount of load. In order to satisfy the given time limit, both CPU and GPU cores have to finish the execution before the given time limit. Now we can rewrite the energy optimization problem with separate CPU and GPU subproblems as follows:

$$\begin{aligned} \text{minimize} \quad & E_{(n_c,f_c,x_c)CPU} + E_{(n_g,f_g,x_g)GPU} \\ \text{subject to} \quad & T_{(n_c,f_c,x_c)CPU} \leq T_1 \\ & T_{(n_g,f_g,x_g)GPU} \leq T_1 \\ & x_c + x_g = 1 \\ & n_c \in N_c, n_g \in N_g, f_c \in F_c, f_g \in F_g. \end{aligned}$$

To find the minimum energy consumption for a given problem and under a given deadline T, one can separately evaluate E_{CPU} and E_{GPU}. To evaluate E_{CPU}, the number of combinations one has to consider are $N_c \cdot F_c \cdot R$, and similarly to evaluate E_{GPU} one has to consider $N_g \cdot F_g \cdot R$ combinations. So, a total of A' evaluations are required where

$$A' = (N_c \cdot F_c + N_g \cdot F_g) \cdot R. \tag{11}$$

We can further reduce the search space by using the observation that energy may be derived from power consumption and execution time on the generic system. So

$$E_{CPU}(n_c, f_c, x_c) = T_{CPU}(n_c, f_c, x_c) \cdot P_{CPU}(n_c, f_c, x_c) \tag{12}$$

$$E_{GPU}(n_g, f_g, x_g) = T_{GPU}(n_g, f_g x_g) \cdot P_{GPU}(n_g, f_g, x_g). \tag{13}$$

From our experiments we found that power consumption on CPU and GPU depends on the clock frequency of the processor and the number of processors used, and does not generally depend on workload. Also, execution time is directly proportional to frequency so, we can write

$$P_{CPU}(n_c, f_c, x_c) \approx P_{CPU}(n_c, f_c) \tag{14}$$

$$T_{CPU}(n_c, f_c, x_c) \approx \frac{f_c max}{f_c} \cdot T_{CPU}(n_c, f_c max, x_c) \tag{15}$$

$$P_{GPU}(n_g, f_g, x_g) \approx P_{GPU}(n_g, f_g) \tag{16}$$

$$T_{GPU}(n_g, f_g, x_g) \approx \frac{f_g max}{f_g} \cdot T_{GPU}(n_g, f_g max, x_g). \tag{17}$$

So practically, by evaluating different numbers of CPU cores and GPUs for the available frequencies, we can derive power consumption for any given load. Thus the number of possible evaluations for determining power consumption is

$$P = N_c \cdot F_c + N_g \cdot F_g \tag{18}$$

where N_c and N_g are cardinalities of the set of CPU cores and GPUs, respectively, while F_c and F_g are, respectively, the number of available CPU and GPU frequencies.

On the other hand, to determine execution time to reconstruct SAR images, we can simply run the CPU workload on n_c CPU cores at the maximum frequency, and the rest of the workload on N_g GPUs also at the maximum frequency. Hence the number of possible evaluations for determining required processing time is

$$T = (N_c + N_g) \cdot R. \tag{19}$$

Given the above discussion, we can derive an energy efficient system configuration by solving the given problem on the system using A'' different evaluations where

$$A'' = P + T = N_c \cdot F_c + N_g \cdot F_g + (N_c + N_g) \cdot R. \tag{20}$$

1) Experimental Results: To demonstrate the usability of our dynamic programming approach, we perform a case study on a heterogeneous platform, for the application SAR image reconstruction. The details of our case study are presented in this section.

We used nine frequency levels as indicated below (in MHz)

$$F_c = \{1200, 1300, 1500, 1700, 1900, 2100, 2200, 2300, 2400\}. \tag{21}$$

Frequency levels used for GPU evaluation is (in MHz)

$$F_g = \{666, 745, 810, 875\}. \tag{22}$$

Our workload distribution set $W = \{\frac{4}{4096}, \frac{8}{4096}, \frac{12}{4096}, \ldots, 1\}$ and Cardinality of W is 1024. We used 5 CPU core combinations, $N_c = \{1, 2, 4, 12, 23\}$ and 1 GPU core. According to the (10) we would have to evaluate $5 \times 1 \times 9 \times 4 \times 1024 = 184,320$ combinations to find the best energy optimum configuration exhaustively. With the approach we used to derive (11), this will be reduced to $(5 \times 9 + 4 \times 1) \cdot 1024 = 50176$ combinations. Furthermore with the method used to derive (20), total evaluations will be reduced to $5 \times 9 + 4 \times 1 + (5 + 1) \cdot 1024 = 6193$.

a) CPU Evaluation: In this section we present results that will validate the assumptions we made to derive (14) and (15).

CPU power P_{CPU} depends on the CPU frequency and the number of CPU cores used. CPU power consumption increases with the number of CPU cores used as well as the increment of the CPU frequency. Theoretically we can represent this relationship as $P_{CPU} \propto f_c \times f_n$. From our experimental results we derived an equation for CPU power as follows using Excel linear regression model:

$$P_{CPU} = 20.67295 + 0.001844 \times n_f \times n_c. \tag{23}$$

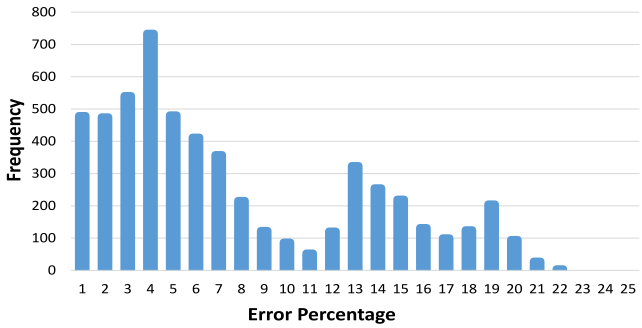


Fig. 18. Frequency of error percentage between actual power and predicted power for CPU using (23) for 5550 test cases.

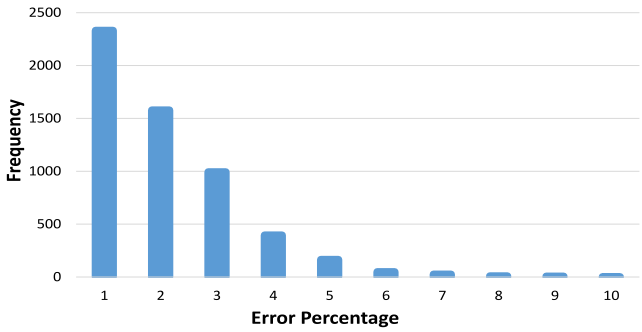


Fig. 19. Frequency of error percentage between actual power and predicted power by using (24) to (28) for 5550 test cases.

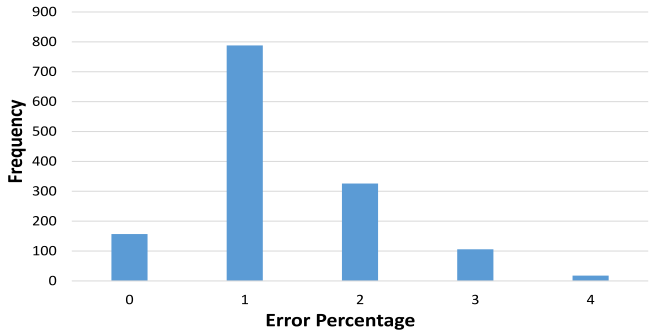


Fig. 20. Frequency of error percentage between actual time and predicted time for CPU evaluation for 1394 test cases.

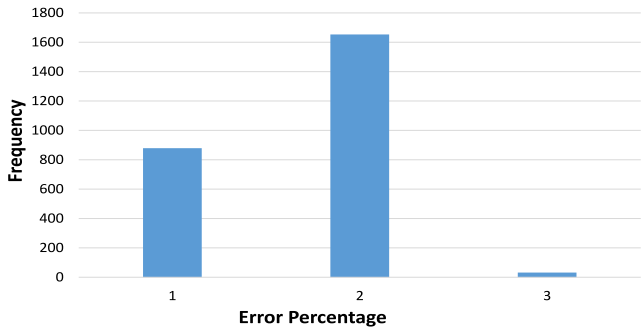


Fig. 21. Frequency of error percentage between actual power and predicted power for GPU evaluation.

We used 5550 evaluations to derive (23). Error percentage between actual power consumption and predicted power consumption for some cases is more than 20% as shown in the Fig. 18.

As the error is higher than an acceptable percentage we derived power equations separately for each core as follows:

$$P_{CPU}(1,f) = 12.49386255 + 0.008055874 \times n_f \quad (24)$$

$$P_{CPU}(2,f) = 9.317893545 + 0.011280284 \times n_f \quad (25)$$

$$P_{CPU}(4,f) = 8.144109452 + 0.014376154 \times n_f \quad (26)$$

$$P_{CPU}(12,f) = 7.572171362 + 0.023166177 \times n_f \quad (27)$$

$$P_{CPU}(23,f) = 21.25981 + 0.044759 \times n_f. \quad (28)$$

Fig. 19 shows the error percentage between actual power and predicted power, calculated using (24) to (28) and projected in to one graph for 5550 test cases. As we can see maximum error percentage is less than 10%.

CPU time, T_{CPU} is inversely proportional to the CPU frequency. Proportionality is removed by introducing a constant as shown by

$$T_{CPU}(n_c, f_c, x_c) = \frac{2400}{f_c} \cdot T_{CPU}(n_c, 2400, x_c). \quad (29)$$

Fig. 20 shows the error percentage between actual time and predicted time for 1394 cases. Error percentage lies in an acceptable range as the maximum error percentage is less than 5%.

b) GPU Evaluation: In this section we present results that will validate the assumptions we made to derive (16) and (17).

We used only one GPU for our experiments. Hence GPU power P_{GPU} depends only on the GPU frequency. From our experimental results we derived an equation for GPU power as follows using Excel linear regression model

$$P_{GPU} = -13.283 + 0.240312 \times f_g. \quad (30)$$

We used 2504 evaluations to derive (30). Error percentage between actual power consumption and predicted power consumption is less than 3% as shown in Fig. 21.

GPU time, T_{GPU} is inversely proportional to the GPU frequency. proportionality is removed by introducing a constant as shown by

$$T_{GPU}(f_c, x_c) = \frac{875}{f_g} \cdot T_{GPU}(875, x_c). \quad (31)$$

Fig. 22 shows the error percentage between actual time and predicted time for 2504 test cases for a single GPU. Maximum error percentage is less than 6%

c) Hybrid Evaluation: We present results from our dynamic programming approach to determine the best energy optimum configuration for our hybrid system. We used CPU only and GPU only evaluations given in Section V-F1 a and Section V-F1 b, to generate the energy and time requirements for the hybrid system.

Fig. 23 shows the total energy against runtime for the estimated results for $4 \times 4k$ image with 5000 pulses. As discussed in

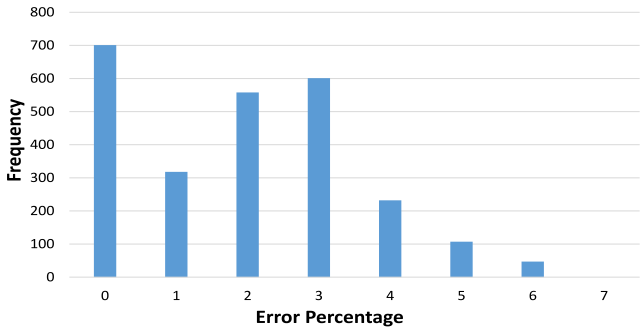


Fig. 22. Frequency of error percentage between actual time and predicted time for GPU evaluation.

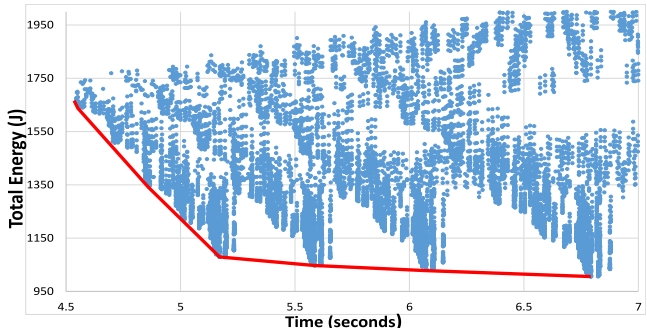


Fig. 23. Energy versus Runtime for the estimated results for $4k \times 4k$ image with 5000 pulses. Pareto-optimal front is denoted by the red curve.

TABLE VI
SET OF CONFIGURATIONS FOR THE POINTS ON THE PARETO CURVE. CPU AND GPU FREQUENCIES ARE IN MHZ

Configuration Number	Number of tiles on CPUs	Number of tiles on GPU	Number of CPUs	CPU Frequency	GPU Frequency
1	548	3548	23	2400	875
2	500	3596	23	2400	875
3	260	3836	12	2400	875
4	136	3960	12	2400	875
5	12	4084	12	2400	875
6	12	4084	12	2400	810
7	12	4084	12	2400	745
8	12	4084	12	2400	666

Section V-F1 a and Section V-F1 b, we used lookup tables which consisted actual evaluations only for the maximum frequencies for CPU and GPU separately to obtain the points on the 23. To calculate the energy consumption for the hybrid system, energy consumed by CPUs and the GPU are summed up.

Table VI shows the set of configurations that lies on the pareto-optimal curve shown in Fig. 23.

We validated energy and runtime for the configurations given in Table VI by executing them on our hybrid platform. Runtime of the estimated results are compared with the actual runtime and the results are shown in Fig. 24. Fig. 25 compares estimated total energy consumption and actual energy consumption for the configurations given in Table VI.

Table VII shows the error percentage between actual and estimated runtime and energy. According to Table VII estimated

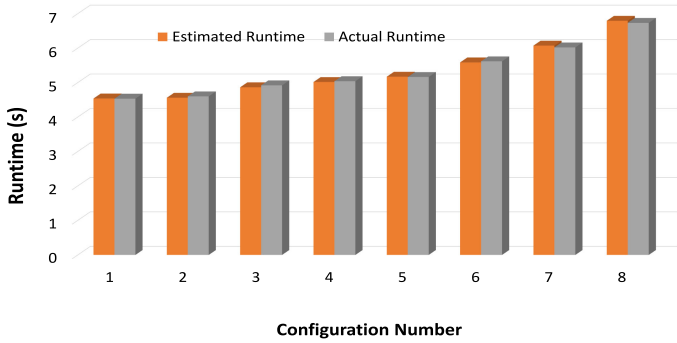


Fig. 24. Comparison of estimated runtime based on our model and the actual runtime for the configurations given in Table VI.

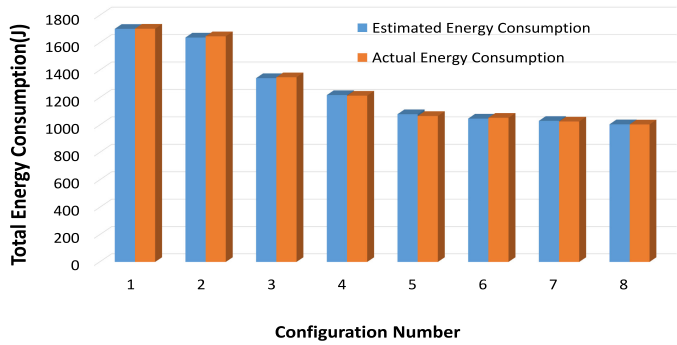


Fig. 25. Comparison of estimated energy based on our model and the actual energy for the configurations given in Table VI.

TABLE VII
ERROR PERCENTAGE IN RUNTIME AND ENERGY BETWEEN ESTIMATED VALUES AND ACTUAL VALUES

Configuration Number	Error % in runtime	Error % in energy
1	-0.104	0.064
2	0.863	0.555
3	1.154	0.467
4	0.472	-0.345
5	-0.158	-1.243
6	0.551	0.550
7	-0.879	-0.421
8	-0.899	-0.101

runtime and energy values are within 2% of the respective actual runtime and energy values.

Fig. 26 gives a graphical representation of the pareto-optimal curve for the estimated values and the actual energy and runtime values for the points in the pareto-optimal curve.

In order to validate the pareto-optimal curve obtained using estimated results, we evaluated over 5000 different configurations in our hybrid platform for 4096×4096 image with 5000 pulses. Fig. 27 shows pareto-optimal curves obtained using estimated results and actual results. According to the pareto-curves we can decide that the pareto-optimal curve obtained using estimated results are almost similar to the pareto-curve of the actual results.

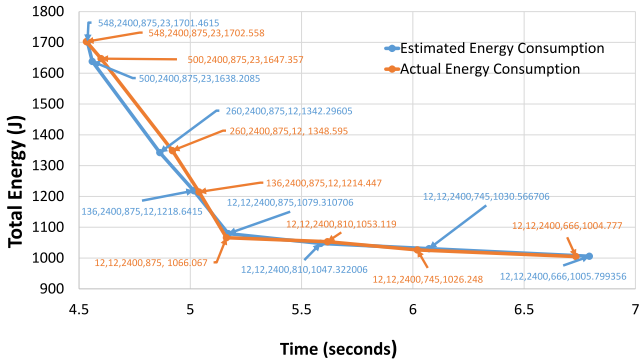


Fig. 26. Estimated and actual energy and runtime values for the points on the pareto-optimal curve. Configuration is given in the order or number of tiles on CPUs, CPU frequency, GPU frequency, number of CPU cores, and energy.

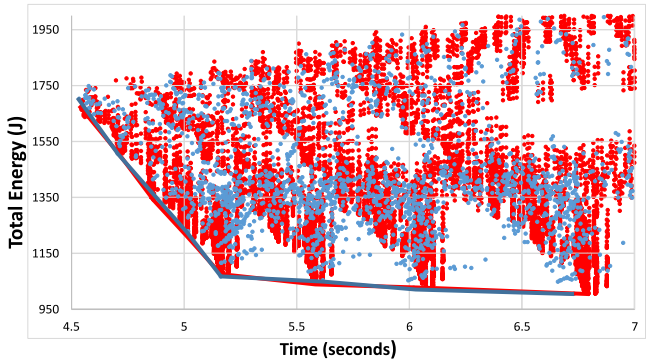


Fig. 27. Separate pareto-optimal curves for the estimated results and the actual results. Red color points and the curve shows estimated values and pareto curve for the estimated results where blue color points and the curve shows actual values and pareto curve for the actual results.

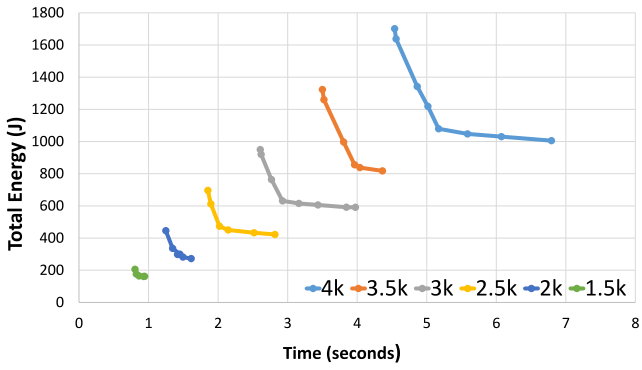


Fig. 28. Pareto-optimal curves for different image sizes with 5000 pulses.

We estimated energy and runtime values for different image sizes such as 4096×4096 (4 k), 3584×3854 (3.5 k), 3072×3072 (3 k), 2560×2560 (2.5 k), 2048×2048 (2 k), and 1536×1536 (1.5 k) and pareto-optimal curves for these image sizes are shown in Fig. 28. As expected energy and runtime decreases for the smaller image sizes while preserving the shape of the curve.

VI. Conclusion

In this article, we discuss methods for SAR image reconstruction on a single GPU, a multicore CPU system, and a CPU–GPU hybrid system. We addressed the work allocation challenges and presented methods to distribute work efficiently between CPU and GPU cores using static work allocation methods. We discussed the problems that arise in hybrid reconstruction when multiple resolution levels are required in the reconstructed image. We generated models to predict time performance and compared the predicted optimum work allocation with the actual optimum allocation concluding that the developed prediction models work well. We investigated how the time performance, power, and energy varies with the use of DVFS on CPU and GPU and presented a pareto curve to obtain tradeoffs between energy and time performance. Furthermore, we developed a model to find the minimum energy configuration when a time constraint is given. The dynamic programming approach we proposed in this article allows to obtain better configurations under the given constraints. This approach drastically reduces the exhaustive search space to find the best configurations while keeping the accuracy of the estimated results to more than of 90%. In future, we will extend our model to multiple GPUs.

References

[1] C. V. Jakowatz, D. E. Wahl, P. H. Eichel, D. C. Ghiglia, and P. A. Thompson, *Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach*. New York, NY, USA: Springer, 2012.

[2] G. C. Walter, S. G. Ron, and M. M. Ronald, *Spotlight Synthetic Aperture Radar, Signal Processing Algorithms*. Boston, MA, USA: Artech House, 1995.

[3] I. G. Cumming and F. H. Wong, *Digital Processing of Synthetic Aperture Radar Data: Algorithms and Implementation*. Norwood, MA, USA: Artech House, 2005.

[4] M. I. Duersch, "Backprojection for synthetic aperture radar," Ph.D. dissertation, Dept. Elect. Comput. Eng., Brigham Young Univ., Provo, UT, USA, 2013.

[5] W. Chapman, S. Ranka, S. Sahni, M. Schmalz, L. Moore, and B. Elton, "A framework for rendering high resolution synthetic aperture radar images on heterogeneous architectures," in *Proc. IEEE Symp. Comput. Commun.*, 2014, pp. 1–6.

[6] A. Wijayasiri, T. Banerjee, S. Ranka, S. Sahni, and M. Schmalz, "Dynamic data driven image reconstruction using multiple GPU," in *Proc. IEEE Int. Symp. Signal Process. Inf. Technol.*, 2016, pp. 241–246.

[7] T. Banerjee, J. Rabb, and S. Ranka, "Performance and energy benchmarking of spectral solvers on hybrid multicore machines," *Sustain. Comput.: Inform. Syst.*, vol. 12, pp. 10–20, 2016.

[8] R. Dolbeau, S. Bihan, and F. Bodin, "HMPP: A hybrid multi-core parallel programming environment," in *Proc. Workshop General Purpose Process. Graph. Process. Units*, vol. 28, 2007.

[9] S. W. Keckler, H. P. Hofstee, and K. Olukotun, *Multicore Processors and Systems*. New York, NY, USA: Springer, 2009.

[10] M. Soumekh, *Synthetic Aperture Radar Signal Processing*. New York, NY, USA: Wiley, 1999, vol. 7.

[11] M. D. Desai and W. K. Jenkins, "Convolution backprojection image reconstruction for spotlight mode synthetic aperture radar," *IEEE Trans. Image Process.*, vol. 1, no. 4, pp. 505–517, Oct. 1992.

[12] L. M. Ulander, H. Hellsten, and G. Stenstrom, "Synthetic-aperture radar processing using fast factorized back-projection," *IEEE Trans. Aerosp. Electron. Syst.*, vol. 39, no. 3, pp. 760–776, Jul. 2003.

[13] L. A. Gorham and L. J. Moore, "SAR image formation toolbox for MATLAB," in *Proc. SPIE Defense, Secur., Sens.*, 2010, pp. 769 906–769 906.

[14] A. Capozzoli, C. Curcio, and A. Liseno, "Fast GPU-based interpolation for sar backprojection," *Prog. Electromagnetics Res.*, vol. 133, pp. 259–283, 2013.

[15] W. Chapman *et al.* "Parallel processing techniques for the processing of synthetic aperture radar data on GPUs," in *Proc. IEEE Int. Symp. Signal Process. Inf. Technol.*, 2011, pp. 573–580.
[16] A. Fasih and T. Hartley, "GPU-accelerated synthetic aperture radar back-projection in cuda," in *Proc. IEEE Radar Conf.*, 2010, pp. 1408–1413.
[17] M. Garland *et al.* "Parallel computing experiences with cuda," *IEEE Micro*, vol. 28, no. 4, pp. 13–27, Jul./Aug. 2008.
[18] J. Li, S. Ranka, and S. Sahni, "GPU matrix multiplication," *Multicore Comput., Algorithms, Architectures, Appl.*, vol. 345, 2013.
[19] CUDA Programming Guide, NVIDIA Developer, Santa Clara, CA, USA. (2016).
[20] C. Lomont, "Introduction to Intel advanced vector extensions," *Intel, Santa Carla, CA, USA,* White Paper, 2011.
[21] A. Tanikawa, K. Yoshikawa, T. Okamoto, and K. Nitadori, "N-body simulation for self-gravitating collisional systems with a new simd instruction set extension to the x86 architecture, advanced vector extensions," *New Astron.*, vol. 17, no. 2, pp. 82–92, 2012.
[22] F. M. M. ul Islam and M. Lin, "Hybrid DVFS scheduling for real-time systems based on reinforcement learning," *IEEE Syst. J.*, vol. 11, no. 2, pp. 931–940, Jun. 2017.
[23] M. K. Bhatti, C. Belleudy, and M. Auguin, "Hybrid power management in real time embedded systems: An interplay of DVFS and DPM techniques," *Real-Time Syst.*, vol. 47, no. 2, pp. 143–162, 2011.
[24] J. M. Rabaey, *Digital Integrated Circuits: A Design Perspective*. Englewood Cliffs, NJ, USA :Prentice-Hall, 1996.
[25] R. Gonzalez, B. M. Gordon, and M. A. Horowitz, "Supply and threshold voltage scaling for low power CMOS," *IEEE J. Solid-State Circuits*, vol. 32, no. 8, pp. 1210–1216, Aug. 1997.
[26] M. Weiser, B. Welch, A. Demers, and S. Shenker, "Scheduling for reduced CPU energy," in *Mobile Computing*. Boston, MA, USA: Springer, 1994, pp. 449–471.
[27] E. Le Sueur, "An analysis of the effectiveness of energy management on modern computer processors," Master's Thesis, *School Comput. Sci. Eng., Univ. New South Wales, Sydney, Australia*, 2011.
[28] E. V. Denardo, *Dynamic Programming: Models and Applications*. Chelmsford, MA, USA: Courier, 2012.
[29] Y. Liu, H. Yang, R. P. Dick, H. Wang, and L. Shang, "Thermal vs energy optimization for DVFS-enabled processors in embedded systems," in *Proc. IEEE 8th Int. Symp. Qual. Electron. Des.*, 2007, pp. 204–209.
[30] S. E. Elmaghraby, "Resource allocation via dynamic programming in activity networks," *Eur. J. Oper. Res.*, vol. 64, no. 2, pp. 199–215, 1993.

Adeesha Wijayasiri received the B.Sc. degree in engineering with a first class honours from the University of Moratuwa, Sri Lanka, in 2013, and the Ph.D. degree from the Department of Computer and Information Science and Engineering (CISE) at the University of Florida, Gainesville, FL, USA, in 2018.

He is a Lecturer with the Department of Computer Science and Engineering at the University of Moratuwa. Prior to joining the University of Florida, he worked as a Contract Lecturer with the Department of Computer Science and Engineering, University of Moratuwa. He was a Volunteer for the Sri Lanka Police for Forensic video analyzing and also worked as a Contractor for International Organization for Migration, Sri Lanka, developing software for analyzing illegal immigrant patterns. His research interests include parallel programming, high performance computing, and image processing.

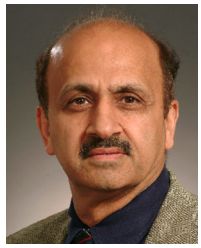
Tania Banerjee received the Ph.D. degree from the Department of Computer and Information Science and Engineering, University of Florida, Gainesville, FL, USA.

She is a Research Assistant Scientist with the Department of Computer and Information Science & Engineering, University of Florida. Her research interests include data structures, sequential and parallel algorithms, high-performance and energy-efficient computing, machine learning, and intelligent transportation systems.

Sanjay Ranka received the B.Tech. degree in computer science from Indian Institute of Technology, Kanpur, India, and the Ph.D. degree in computer science from the University of Minnesota, Minneapolis, MN, USA.

He is a Professor with the Department of Computer Information Science and Engineering at University of Florida, Gainesville, FL, USA. From 1999–2002, he was the Chief Technology Officer with Paramark, Sunnyvale, CA, USA. At Paramark, he developed a real-time optimization service called PILOT for marketing campaigns. He has coauthored four books, 290+ journal, and refereed conference articles. His current research interests are high performance computing and big data science with a focus on applications in CFD, health care, and transportation.

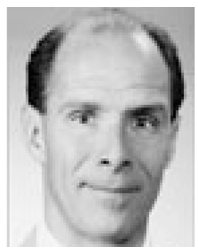
Dr. Ranka is a fellow of the AAAS and a past member of IFIP Committee on System Modeling and Optimization. He is an Associate Editor-in-Chief for the *Journal of Parallel and Distributed Computing* and an Associate Editor for *ACM Computing Surveys*, *Applied Sciences*, *Applied Intelligence*, IEEE/ACM TRANSACTIONS ON COMPUTATIONAL BIOLOGY AND BIOINFORMATICS, *Sustainable Computing: Systems and Informatics*, *Knowledge and Information Systems*, and *International Journal of Computing*.



Sartaj Sahni received the B.Tech. degree in electrical engineering from the Indian Institute of Technology, Kanpur, Kanpur, India, and the Ph.D. degree in computer science from Cornell University, Ithaca, NY, USA.

He is a Distinguished Professor of Computer and Information Sciences and Engineering with the University of Florida, Gainesville, FL, USA. He has authored or coauthored over 400 research papers and written 15 texts. He holds 15 US patents. His research interests include the design and analysis of efficient algorithms, parallel computing, interconnection networks, design automation, and medical algorithms.

Dr. Sahni is a member of the European Academy of Sciences, a Fellow of ACM, AAAS, and Minnesota Supercomputer Institute, and a Distinguished Alumnus of the Indian Institute of Technology, Kanpur, India. He is also a member of the steering committee of several international conferences. In 1997, he was the recipient of the IEEE Computer Society Taylor L. Booth Education Award "for contributions to Computer Science and Engineering education in the areas of data structures, algorithms, and parallel algorithms," and in 2003, he was the recipient of the IEEE Computer Society W. Wallace McDowell Award "for contributions to the theory of NP-hard and NP-complete problems." He was the recipient of the 2003 ACM Karl Karlstrom Outstanding Educator Award for "outstanding contributions to computing education through inspired teaching, development of courses and curricula for distance education, contributions to professional societies, and authoring significant textbooks in several areas including discrete mathematics, data structures, algorithms, and parallel and distributed computing." In 2016, he was the recipient of the IEEE Technical Committee on Scalabale Computing (TCSC) Award for Excellence in Scalable Computing for "fundamental contributions to scalable computing and leadership in service to the scalable computing community." He was an Editor-in-Chief for *ACM Computing Surveys*, Co-Editor-in-Chief for the *Journal of Parallel and Distributed Computing*, and is currently on the editorial board of 17 journals.



Mark Schmalz received the O.D. degree in optometry and the Ph.D. degree in computer science from the University of Florida, Gainesville, FL, USA.

He is an Associate Scientist with the Department of Computer and Information Science and Engineering at University of Florida. He has authored or coauthored over 170 book chapters or research papers in open conference proceedings and journals, edited over 15 books. His research interests are in high-performance parallel computing, image and signal processing, human and machine vision, automated processing and understanding of digital imagery, data compression, and cryptology. He is affiliated with UF's Digital Arts and Sciences program, with research interests in psychoacoustics and computer-generated music.