Evaluation of an Accelerator Architecture for Speckle Reducing Anisotropic Diffusion

Siddharth Nilakantan
Drexel University
Philadelphia, PA, USA.
sn446@drexel.edu

Srikanth Annangi
Drexel University
Philadelphia, PA, USA.
sa662@drexel.edu

Nikhil Gulati
Drexel University
Philadelphia, PA, USA.
ng54@drexel.edu

Karthik Sangaiah
Drexel University
Philadelphia, PA, USA.
ks499@drexel.edu

Mark Hempstead
Drexel University
Philadelphia, PA, USA.
mdh77@drexel.edu

ABSTRACT

Increasing chip power density has brought application specific accelerator architectures to the forefront as an energy and area efficient solution. While GPGPU systems take advantage of specialized hardware to perform computationally intensive tasks faster than chip multiprocessor (CMP) systems, accelerators are hardware units that are designed to execute a specific application efficiently. Real-time ultrasound imaging applications require the removal of multiplicative noise while maintaining a steady frame-rate, and are good candidates to explore accelerator-based systems. In this paper, we propose and evaluate the architecture of an accelerator designed to improve performance of SRAD image enhancing algorithm. We compare the projected performance of the SRAD accelerator to software implementations on a multi-core CPU and a CPU+GPU system. The proposed architecture achieves higher throughput by eliminating redundant fetches from memory and by storing intermediate data locally. The speedup of the GPU is found to be 3.2x over the CPU, while the accelerator achieved a speedup of 24x. The area efficiency of the GPU and accelerator is up to 1.6x and 370x better than the CPU, respectively. In comparison with the CPU, we find that the energy consumed for operation on a single frame is found to be 1.5x lesser on the GPU and up to 580x lesser on the accelerator.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Data-flow architectures; C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms

Algorithms, Performance, Design

Keywords

SRAD, Accelerator, Performance, GPU

1. INTRODUCTION

It has been widely observed that the microprocessor industry has shifted from architectures with complex single cores optimized for high sequential performance to architectures with many cores on a single chip. This shift is due to increasing power demands that come with technology scaling that cannot be met cost effectively by current cooling and battery technology [13, 17]. Thus, tight power budgets on systems have driven the industry to use multiple identical power-efficient cores within a given die area [3, 9].

Heterogeneous systems make use of specialized hardware, such as GPUs, to deliver higher performance and maintain the same or better power efficiency when compared to homogeneous systems [8, 12]. Another area of research currently being pursued by computer architects and chip designers is accelerator-based systems where highly efficient application specific hardware units are organized together on a single chip catered towards a specific application domain [1, 5]. Many embedded systems-on-chip (SoCs) (e.g. [2]) in use today employ specialized hardware components.

A variety of medical imaging applications (such as portable and handheld ultrasound) require real-time processing of images in the field to enable immediate review of results. The need for speed and efficiency by these image processing applications make them good candidates for accelerator-based systems. In this paper, we describe the architecture of an accelerator targeted at a particular ultrasound image enhancing algorithm known as 'Speckle Reducing Anisotropic Diffusion'. The accelerator control logic is customized to the data flow and computational requirements of the algorithm. Each computation in SRAD is complex and includes input from all neighboring pixels, and consequently, we have found it difficult to obtain good performance using a general purpose solution. The motivating case for this work is a portable low power device that could run SRAD in real time on a high-resolution display. To achieve this goal, a high performance, low cost SRAD accelerator could be embedded into a portable ultrasound device to improve visual quality of the image.

Our contributions in this paper are as follows:

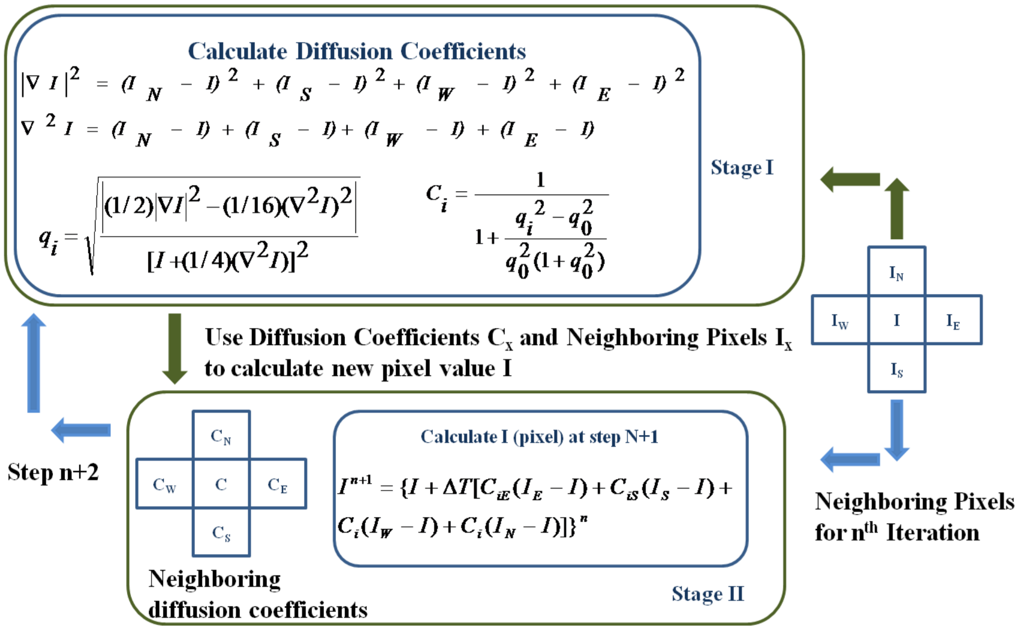
Calculate Diffusion Coefficients
$|\nabla I|^2 = (I_N - I)^2 + (I_S - I)^2 + (I_W - I)^2 + (I_E - I)^2$
$\nabla^2 I = (I_N - I) + (I_S - I) + (I_W - I) + (I_E - I)$
$q_i = \sqrt{\frac{(1/2)|\nabla I|^2 - (1/16)(\nabla^2 I)^2}{[I + (1/4)(\nabla^2 I)]^2}}$
$C_i = \frac{1}{1 + \frac{q_i^2 - q_0^2}{q_0^2(1 + q_0^2)}}$
Stage I
I_N
I_W
I
I_E
I_S
Use Diffusion Coefficients C_x and Neighboring Pixels I_x to calculate new pixel value I
Step n+2
C_N
C_W
C
C_E
C_S
Calculate I (pixel) at step N+1
$I^{n+1} = \{I + \Delta T[C_{iE}(I_E - I) + C_{iS}(I_S - I) + C_i(I_W - I) + C_i(I_N - I)]\}^n$
Neighboring diffusion coefficients
Stage II
Neighboring Pixels for nth Iteration

Figure 1: SRAD Data Flow

- We propose a novel scalable accelerator architecture targeted towards the SRAD application.
- We characterize the application behavior on three platforms, namely a CPU, GPU, and accelerator based system.
- We compare the platforms (CPU, GPU, Accelerator) and quantify where the bottlenecks in each platform lie, as well as how they might be mitigated.
- We qualify the accelerator architecture in terms of System on Chip requirements to conclude that it will be a highly effective solution for a SoC based design in medical imaging applications.

The rest of this paper is organized as follows. A detailed description of the selected SRAD algorithm is provided in Section 2. Related work in this field is shown in Section 3. We describe the SRAD accelerator architecture in detail in Section 4. The experimental setup and methodology for our experiment is shown in Section 5. Comparison and results are presented in Section 6. Discussion of results are presented in 7, and we conclude in Section 8.

2. SPECKLE REDUCING ANISOTROPIC DIFFUSION (SRAD)

2.1 Algorithm Description

Speckle is a form of multiplicative, locally correlated noise that is present in imaging applications, such as medical ultrasound imaging. For images that contain speckle, any enhancement technique must aim to remove the speckle without destroying important image features. SRAD is an adaptive approach for speckle removal widely used in radar and medical ultrasound imaging. Yu et al. [24] show that in the presence of speckle noise, SRAD excels beyond the traditional speckle removal filters and the conventional anisotropic diffusion method with respect to the metrics of mean preservation, variance reduction, and edge localization.

2.2 Algorithm Implementation

SRAD can be classified as an application with data organized in a structured grid, similar to many other benchmarks in the Rodinia Benchmark Suite [4]. Thus, SRAD is a good representative application of a larger class of applications that we evaluate as a potential target for accelerator designs and accelerator-based systems.

Yu and Acton [24] describe a discrete form of the algorithm kernel which can be adapted to software and hardware. Although this work describes the SRAD computation in three functional phases, we found only two distinct computational stages upon analysis of the algorithm. The complexity of the computations involved in solving the equations poses the challenge in customizing the pipeline towards the algorithm. Besides the two core kernel computations, SRAD includes a small serial portion that calculates the mean and variance over the entire image and uses that to calculate q_0, the speckle coefficient of variation. Previous work has assumed the value of q_0 to stay constant for a particular image, and in fact, it can be assumed to stay constant for a particular video sequence [22].

Fig. 1 shows the stages and the operations performed in each stage. In the first stage (Stage 1), the directional derivatives for each pixel are determined, followed by the calculation of a diffusion coefficient C_i corresponding to each pixel in the input image. Stage 1 thus requires immediate neighboring pixel values to be available for each pixel currently being processed. The second stage (Stage 2) calculates the divergence of the diffusion coefficients multiplied by the directional derivatives computed in the Stage 1 and uses the divergence to compute the new pixel value. For each new pixel value computed, Stage 2 requires immediate neighboring pixel values, and also requires immediate neighboring diffusion coefficient values computed in Stage 1 as well. The computation for each active pixel I in both stages is dependent on its adjacent pixels (IE, IW, IN, IS), both in the row and column directions. Assuming row by row processing of the input image, the computations in Stage 2 depend on previous, current and future results of Stage 1. This makes any potential parallel optimization of the algorithm more complex. The algorithm is run for a number of iterations which, in most cases, is empirically determined by the user; an unnecessary number of iterations increases the computation time and produces very little additional image quality.

We used the parallel implementations of the SRAD algorithm for multi-core CPU and GPU platforms from the Rodinia Benchmark Suite [4]. Due to the dependence between the two stages, each stage is a separate kernel that successively operates on the entire image. We propose a custom hardware implementation, where the two stages are pipelined and operate on different rows of the image simultaneously. We show that our pipelined approach improves throughput and provides higher efficiency than the other platforms.

3. RELATED WORK

The field of building application specific architectures has seen a lot of growth in recent years, with plenty of ASIC designs for existing and upcoming applications [14, 15, 18]. The Medical Imaging industry continues to thrive [6] upon recent advances in faster and more efficient custom hardware architectures that can be used for different imaging modalities. Wu et al. implement SRAD on an FPGA using a modified version of the algorithm [22]. In this work, a single module is implemented which computes Stage 1 and 2. The computations to be performed in Stage 1 are simplified to reduce complexity and the divergence computation in Stage 2 is done over many cycles with partial sums being computed when neighboring diffusion coefficients are available. Our work extends the work of Wu et al. by modifying the architecture to be 1) scalable with intra-core parallelism, 2) efficient in terms of area, power and bandwidth by storing only a part of the image in the dedicated SRAM and 3) of higher precision resulting in better dynamic range. We have also drawn comparisons with other platforms so as to evaluate the trade-offs in a system that employs the SRAD algorithm.

Though traditional ASICs have been used successfully, industry has also tried to look towards using general-purpose processors for specific applications. Though they are cheaper, they do not give as much performance as ASICs and also consume more power. Domain specific accelerators on the other hand can achieve the performance of an ASIC and can be integrated with simple cores and achieve reduced power density to strike a performance/power balance [7]. Cong et. al [6] studies how customized hardware and software can be targeted for the medical imaging domain while maintaining performance/power efficiency.

4. SRAD ACCELERATOR ARCHITECTURE

4.1 Baseline Architecture

The baseline architecture is composed of a Custom Processing Element (CPE) and a group of FIFOs. The FIFO that sends and receives data from the external system is large enough to hold a portion of the image and is hence called the image buffer (FIFO - IB). All the FIFOs are implemented as SRAM memories with FIFO control logic which generates the appropriate addresses. Computations in Stage 1 and 2 are assigned to Logic Element 1 and 2 respectively. Figure. 2 shows the diagram of the baseline accelerator architecture. Each Logic Element derives its inputs from a 3×3 Register Array (Register Array 1 and 2). The center or middle pixel in the Register Array (non-highlighted in the diagram) is the active pixel for computation in a particular clock cycle, so both Logic Elements 1 and 2 operate on the middle data elements of their corresponding arrays.

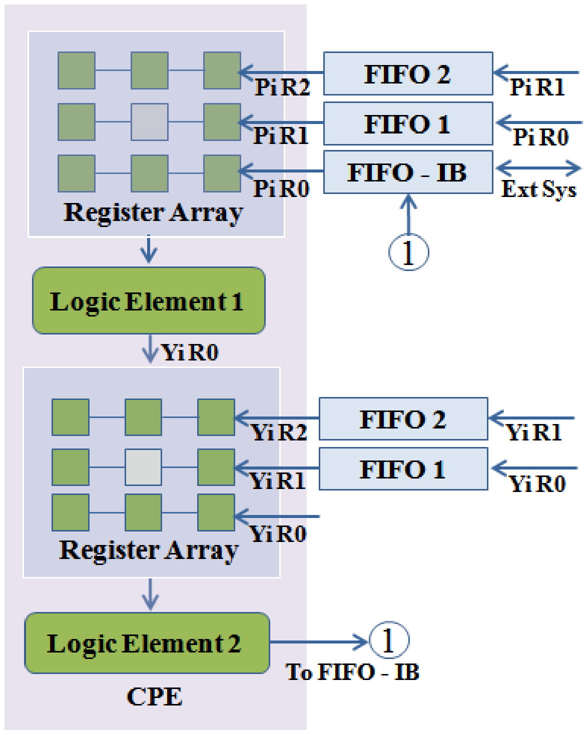

Figure 2: CPE Architecture

The FIFOs produce pixels from three different rows which are fed into one of the corresponding rows in Register Array 1. The pixel data flow in the Register Array matrix on two successive clocks is as follows: In the first cycle, the center pixel of the array along with it's neighbors, $(IN, IE, IWandIS$ from Figure 1) are fed into Logic Element 1. At the end of the cycle, all the pixels in the array are shifted to the left, and a new pixel is fetched for each row from the FIFOs into the rightmost locations in the array. Thus, in the next cycle, the new center pixel, which corresponds to IE in the previous cycle, is processed. In this manner, over a number of cycles, the entire center row of pixels is processed, while using the rows above and below as neighbors.

Every clock, the FIFOs feeding every row also write the same data to the FIFO immediately above them. Thus, when the entire center row has been processed by Logic Element 1, the same row will then be available in the immediately upper FIFO (topmost FIFO) to act as a neighbor to the next row in the center (next row of the image). Any row in the center FIFO would have previously resided in the FIFO immediately below it (bottom-most FIFO) to act as a neighbor to the previous center row. This flow of data can be tracked from Figure. 2 with PiRx and YiRx ($wherex = 0, 1, 2$). The bottom-most FIFO is the image buffer which receives input data for rows of the image from the external system.

The operations for Logic Element 2 follow almost the same pattern discussed for Logic Element 1 with a few exceptions. Register Array 2 and its corresponding FIFOs also hold diffusion coefficient values as opposed to Register Array 1 which holds only pixel values. Logic Element 2 recomputes the directional derivatives from the pixel values instead of using Register Array 2 and the FIFOs to store all directional derivatives computed by Logic Element1, as this solution consumes lesser area. Logic Element 2 produces new pixel values which are dumped back into the image buffer as shown in Figure. 2. Note that Logic Element 2, which also operates on the center data elements of Register Array 2, needs the neighbor values of both pixels and diffusion coefficients. Each clock period, Logic Element 1 provides one coefficient and one pixel which are stored in Register Array 2 in the bottom-most row. The flow of data in Register Array 2 and its FIFOs follow the same pattern as in the first stage, except that the input row is fed from the output of Logic Element 1 as opposed to the image buffer.

4.2 Scaling the Architecture for Parallelism

This section explores the method of scaling the architecture described in the previous section. In Section 4.1, the combined pattern of dataflow in the Register Arrays and FIFOs ensure there are no redundant pixel fetches from memory. Also, the FIFOs in Stage 2 enable computation to begin in Stage 2 after a latency equal to the number of cycles it takes to shift one entire row up, instead of waiting for the entire image to be processed in Stage 1 as found in the CPU and GPU implementations. The overhead for these gains are the costs of fetching and storing two extra rows in both stages. The scaling described in this section takes advantage of the parallelism between the required processing of different pixels and amortizes the fetch and storage overhead over more number of computations.

The architecture can be scaled to take advantage of parallelism by using an array of Logic Elements in each stage instead of just one. Figure. 3 shows the architecture with an array of 3×3 Logic Elements. If the array of Logic Elements is sized $n \times n$, then the Register Array will be sized $(n + 2) \times (n + 2)$ as there will be two extra rows of storage with FIFOs and two extra columns in the Register Array that provide neighbor information for the pixels at the edge of the array. The $n \times n$ array of Logic Elements in each stage compute the result for the $n \times n$ array of pixels in the center of the Register Array. The flow of data in the two stages is very similar to that described previously, except that every operation is over n pixels as opposed to 1 pixel. During each clock period, n pixels are fetched from the image buffer and stored in Register Array 1. The previous contents of the Register Array 1 are shifted by n. Register Array 2 receives $n \times n$ pixels and $n \times n$ diffusion coefficients during each clock with FIFOs housing two extra rows of pixels and diffusion coefficients. The image buffer will have n banks to provide data from n rows (FIFO IB 0, 1 and 2 in the diagram). Hereafter n is also referred to as the CPE dimension.

4.3 Estimation of scaled CPE Metrics

In this section, we investigate the impact of scaling the design by sweeping the number of Logic Elements (n^2) in a stage. This refers to the number of Logic Elements in both stages as the size of the array is symmetrical. Henceforth, we refer to Logic Elements as LEs for convenience.

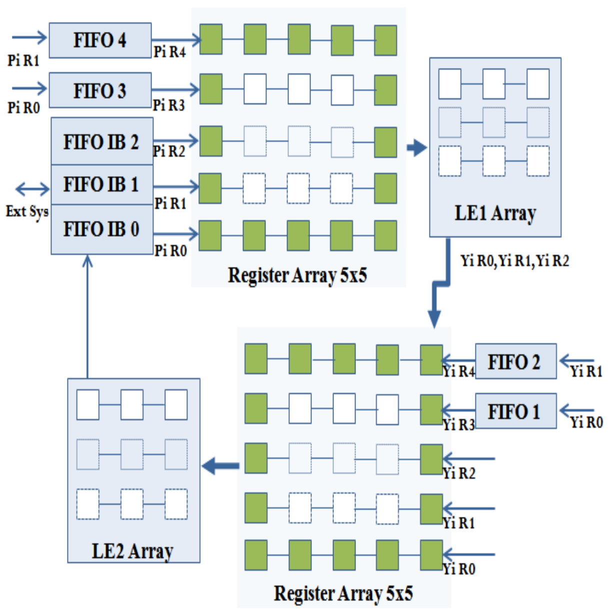

Figure 3: Scaled CPE Architecture

4.3.1 *Projected CPE requirements*

We developed synthesizable RTL for the accelerator (Verilog) with fixed point computation assuming 3 bytes/pixel storage. The fixed point design does not truncate results except for very long divide operations. Logic synthesis was performed with Synopsys Design Compiler targeting the SAED 90nm standard cell library to obtain area and timing estimates for the CPE, assuming typical operating conditions. Our average power estimation is obtained from Synopsys PrimeTime PX using the vector free power analysis flow, with switching activity derived from the range of values used by the SRAD algorithm. The area numbers for the FIFOs were obtained from [19].

The area of the LEs occupy up to 70% of the overall area and the total memory area increases linearly with the number of LEs. The memory bandwidth requirements of a scaled CPE will be directly proportional to n^2 (number of LEs) as it operates on n data elements from n rows each clock. The total average power is also seen to be linear with the number of LEs. Thus, scaling up the LEs will not cause disadvantageous increases in any of these requirements.

4.3.2 *Projected CPE Performance*

We obtained clock cycle time post synthesis for a particular CPE architecture and the FIFOs given the value of n. As the throughput of the CPE per cycle will be n^2 pixels, using the cycle time, we determined the throughput of the device in terms of pixels/second. All Logic Elements and Register Arrays operate in lock-step, which means that either the whole CPE is stalled or it always operates at maximum throughput. Since, the throughput per LE is always the same and each LE processes one pixel per cycle, CPE throughput increases linearly with the number of LEs.

4.3.3 *System level considerations*

As mentioned earlier, our design has two advantages. It takes advantage of intra-core parallelism, as mentioned by Wu et al., to scale up the number of LEs and hence improve throughput. Second, the CPE can perform operations on a portion of the image independent of the rest of the image. This is because no operations need to be performed on the entire image in between iterations. Based on this insight, we can keep the image buffer small to reduce area and store only a portion of the image at one time. This would require two extra rows to be fetched for each portion of the image to provide neighbor information for the first and last rows. The specified number of iterations is completed on that portion of the image before switching in the next portion of the image. However, keeping the image buffer too small would cause more service requests to the external system and increase the overhead of fetching extra rows. Figure. 4 shows the trade-offs in choosing the size of the image buffer for a 9 LE CPE. We assume an embedded system with 32 bit data widths and a memory transfer rate of 2 GB/s. Current generation LPDDR2 memories are capable of providing up to 4 times this bandwidth, but we assume 25% will be deliverable. The overhead of transferring extra rows (shown as a percentage of the frame size) and the number of requests to the external system (shown as percentage of maximum requests) decrease with the size of the image buffer. However, the area of the memory begins to exceed the area of the LE with image buffer size greater than 192KB. It can be seen that the optimum point lies in between the 48KB and 96KB buffer sizes. For our evaluation, we chose 96kB as that represents about 30% of LE area, while sending only 16 transfer requests per frame to the external system. This will be implemented as two memories of 48kB each. Each 48kB memory will have n banks, where n is the CPE dimension. One memory is read from and written to by the CPE, while the other memory is read out and receives a fresh portion of the image from the external system. Once the portion of the image has finished processing, the CPE begins reading from the other memory and vice versa. As we find that the active power of the image buffer lies between 0.1 and 0.15W, which is a very small percentage of overall power, it is not factored in to select the image buffer size. In our evaluation, the maximum image size in the system is 512 x 512. Given this criterion, each input FIFO which houses extra rows has 512 entries of 3 bytes each (3 bytes/pixel). The image buffer, and FIFOs are all clocked at twice the frequency of the CPE, so that it may perform one read and one write into the image buffer in one CPE clock. The access time and cycle times of the memory models generated by CACTI justify this possibility as the CPE is clocked at a low frequency of 91 MHz.

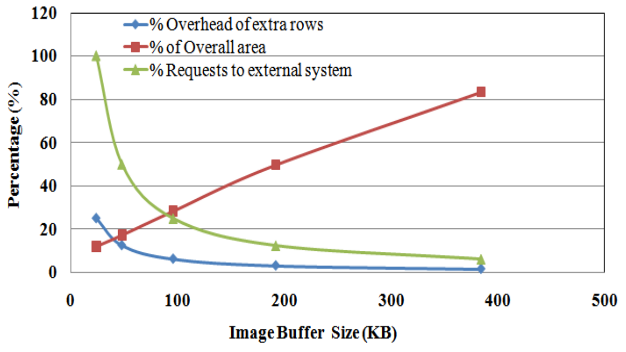

Figure 4: Trade offs for Image Buffer Size

The transfer of data between the external system and the image buffer is completed during the same time window in which the portion of the frame in the image buffer is being processed for all iterations. When a transfer request is processed by the system during this window, it will have the peak transfer rate of the system (2GB/s). Thus, the image transfer will be completed in a very small fraction of time (0.75%). The required average external memory bandwidth in the window is as low as 175MB/s.

5. EXPERIMENTAL SETUP & METHODOLOGY

We evaluate the accelerator architecture for CPE dimensions $n = 1$, 2, 3 and 4. We estimate the performance of the accelerator in frames/sec and compare it to the results obtained by running the software implementations on a GPU and CPU. As implementations of the SRAD algorithms for embedded platforms were unavailable to us, we used desktop versions of the same. Table 1 summarizes the tools and hardware we used for the experiments.

We perform our experiments on three different image sizes: 128×128, 256×256 and 512×512 as these represent the image sizes typically seen in real-time ultrasound applications. Related work on real time ultrasound imaging use 256×512 sized images but do not explore speckle reduction [23]. Prior work on speckle reducing algorithms, including the work on developing the SRAD algorithm use up to 200×200 sized images [24, 21]. We have added the 512×512 image size as a representation of potential resolution improvements in upcoming portable ultrasound devices. The real-time performance requirements of an ultrasound device are at least 30 fps as it is the minimum required to avoid flicker on the display for human eye. As the number of iterations is determined empirically by the user's experience there is no an optimal value to use. However, previous work has used between 50 and 300 iterations [24, 21]. We evaluate our architecture for 50, 100 and 150 iterations of the algorithm. For all platforms we compare only kernel execution time, as the data transfer operation is platform dependent and would be replaced by a data transfer path common to all solutions for SRAD in an embedded SoC. We also do not include the time required to set up the data structures for operation as that is a very small percentage of the total compute time on both the CPU and GPU systems (approx. $1\% - 2\%$). We focused only on the power consumed by the chip, because as mentioned earlier, the data transfer mechanism would be

Table 1: Hardware setup, profiling and design tools

	GPU	CPU1	CPU2	CPE
Hardware Platform	Nvidia 8800GTX	Intel Core i5-2500K	Intel Xeon E5620	NA
Profiling Tool	CUDA Profiler	-	PAPI	VCS HDL Simulator
Design Tools	NA	NA	NA	Design Compiler
Power Measurement	Agilent Multimeter	Intel Power Gadget	-	PrimeTime PX

common to all solutions in the embedded system. To calculate power consumption we measured the current on the $+12V$ power lines on our Desktop system. To facilitate measurement of current on the PCI-e bus for the GPU, we used a riser cable and a Hall-effect current clamp connected to a multi-meter. For the CPU, we used the Intel Power Gadget widget to measure the power utilized by the CPU while computing SRAD.

5.1 CPU Setup

The OpenMP implementation of the SRAD benchmark was executed on a Core i5-2500K Sandy Bridge CPU with 4-way hardware multi-threading for CPU performance and power consumption. Although this CPU was implemented in $32nm$, we use it directly for comparison as the performance and efficiency data shows that even a state of the art CPU is unable to keep up with the other platforms. We used the Performance Application Programming Interface (PAPI) tool [20] to access the hardware performance counters on an Intel Xeon E5620 CPU as PAPI did not support the required counters on the Core i5 Sandy Bridge CPU. These counters accumulate counts on various memory and performance affecting events. Using PAPI, the following metrics were obtained: L1 data cache accesses, L3 data cache accesses, total execution cycles, and stall cycles.

5.2 GPU Setup

As shown in 1, we evaluated the performance of SRAD on a GPU using an Nvidia 8800 GTX, which has 16 streaming multiprocessors (SM), and each SM has 8 streaming processor (SP) cores for a total of 128 cores. The 8800 GTX has 16 KB of on-chip shared memory for each SM, and all SMs have access to 768 MB of off-chip global memory. We used the CUDA visual profiler to gather performance metrics such as kernel execution time and memory bandwidth.

6. PERFORMANCE RESULTS AND COMPARISON

We measure the performance of both the GPU and CPU implementations of the SRAD application and quantify any potential bottlenecks; finally, we present a comparison. We use a maximum image size of 512 × 512.

6.1 Performance Results on each platform

6.1.1 *CPU Performance Results*

We measured the frame rate of the kernel as the number of threads was incremented from 1 to 4 threads. As shown in 5, the frame rate increases as the number of threads, since additional computational units were available to process the kernel in parallel.

The CPU is able to meet the deadline of 30 FPS only for the 128 × 128 image size. This shows that the CPU will

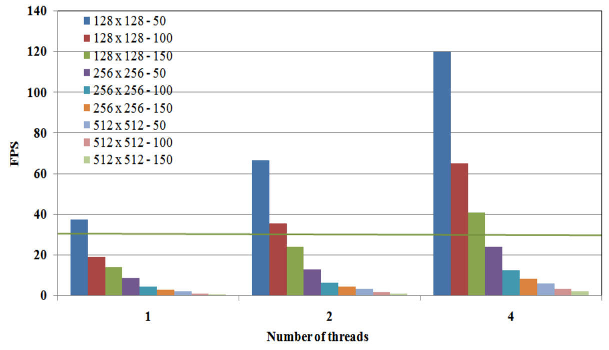

Figure 5: Kernel Performance in fps

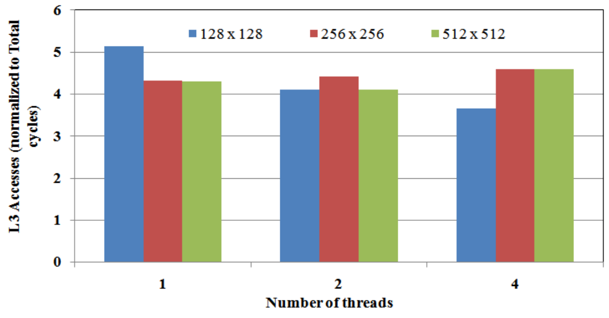

Figure 6: L3 Data Cache Accesses

not meet the real time requirements of the SRAD application with the exception of really small data sizes. Figure. 6 shows the average L3 data cache accesses normalized to the number of execution cycles. The number of L3 cache accesses per cycle per thread remains the same for different number of threads and data sizes since the number of issued operations per cycle to the CPU remains the same, even though data size increases. Although the number of issued operations per cycle is fixed, it is also subject to various stalls and differing memory hierarchy behavior, both of which introduce some variability and cause the CPU memory usage to vary slightly with data size. The results were obtained from using an Intel Xeon E5620 processor also implemented in 32nm, as PAPI was unable to query the performance counters on the core i5 that we use for performance and power comparisons. The theoretical peak of 1 memory access per cycle is not achievable even if the CPU compute time were to be improved, as there will be other sequential instructions that fall in between memory accesses. This implies that for SRAD and similar applications, the per-core cache for the CPU is under-utilized. If more number of computation units could be employed to access a single cache in parallel, there might be better utilization. Also, with each core being optimized for Instruction Level Parallelism (ILP), the cores are not well suited to handle massive Data Level Parallelism (DLP). The proposed improvements to the CPU architecture, namely, more computational units sharing memory and pipelines optimized for DLP, take us naturally toward a GPGPU architecture.

6.1.2 GPU Performance Results

We measured the frame rate of the kernel by incrementally increasing the number of threads per block. Setting the number of threads per block directly impacts the number of active blocks the GPU can schedule, given the available shared memory on the GPU. This eventually limits the number of active threads the GPU can schedule per SM.

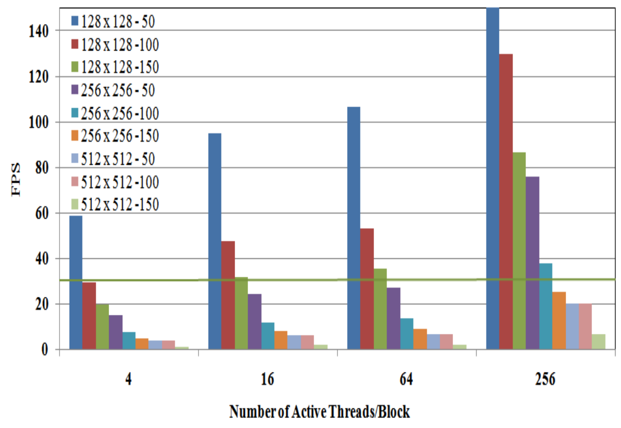

Figure 7: Kernel Performance in fps

As seen in Figure. 7, the GPU does not meet the real time requirements of the application as the frame rate of 30 FPS is not met for the 512 × 512 image size. An approximately linear increase in frame rate is observed until the point where only 2 active blocks can be scheduled per SM (data point 256 threads/block in Figure. 7) with a total of 512 active threads per SM. This is because, even with a maximum of 768 threads available for each SM, the GPU was able to schedule only 512 active threads because it is bounded by 16 KB of shared memory and can accommodate only 2 blocks per SM with each block using 6144 bytes of shared memory. The GPU would thus benefit from more shared memory or a mechanism which would allow more frugal use of the shared memory. Section 6.2.2 analyzes and elaborates on some possible optimizations.

6.1.3 Accelerator Performance Results

We measure the frame rate for the three different image sizes and number of iterations, as a function of the number of LEs. As shown in Figure. 8 the frame rate increases linearly with the number of logic elements for all the image sizes (shown in logarithmic scale as some data points would go off the chart otherwise). The frame rate is directly proportional to the number of LEs. Since the area and bandwidth requirements are almost linear as well, the theoretical maximum speedup and efficiency is limited only by the number of LEs. As shown in Figure. 8, the accelerator system almost meets the real time requirements of the application. On a 9 LE CPE, the frame rate of 30 FPS is met for almost all image sizes except the 512 x 512 frame with 150 iterations. The 16 element CPE meets the required frame rate for all cases. As a result, we will evaluate speedup and efficiency for the 9 and 16 LE CPEs.

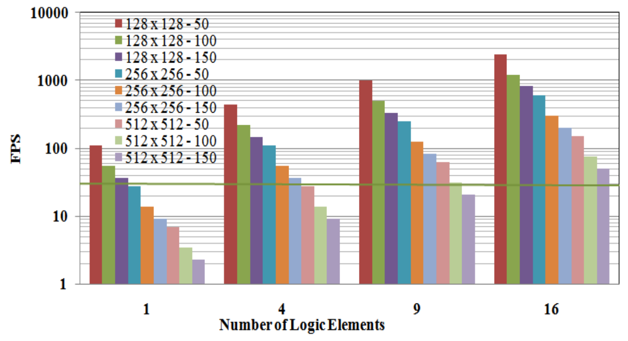

Figure 8: Kernel Performance in fps

6.2 Comparison of Platforms

We compare the platforms on the basis of performance, power, area, and bandwidth requirements. We also focus one section on bandwidth since it is an increasingly important criterion for embedded systems. We define two different types of memory bandwidth to compare the three different SRAD implementations. This distinction is to isolate the bandwidth requirements at different levels of the memory hierarchy.

1. **Internal memory bandwidth:** This is represented by the L1 data cache in the CPU, the shared memory in the GPU and the FIFOs in the CPE. The internal bandwidth calculated is per-core for the CPU, per-SM for the GPU, and per-LE for the CPE as this bandwidth can be scaled based on the number of these respective structures available on chip.

2. **External memory bandwidth:** For the CPU, we define this as any L3 data cache traffic, although it is implemented on-chip. This is justified with two reasons: 1) L2 and L3 caches are usually not present in embedded cores. 2) The L3 in this case is big enough to house the entire image and has a big access penalty. On the GPU, external memory is represented by the off chip global memory, and for the CPE it is assumed to be off-chip DRAM as well. The external bandwidth utilization depends on the number of cores sharing the external memory and accessing it in parallel. We obtain L3 cache bandwidth numbers based on the specifications of the Intel Core i5-2500K.

The CPU bandwidth numbers were derived from metrics observed in PAPI. The internal memory bandwidth utilization for the CPU was calculated using L1 accesses/Total execution cycles as we assume the maximum possible bandwidth is assumed to be 1 memory access/cycle. The external memory bandwidth usage was measured using L3 accesses/Total execution cycles. We obtained the cycle time and the cache line size (64 bytes) from datasheets and used them to calculate bandwidth in Bytes/second [11]. (We have assumed here that each access maximizes utilization of the data bus in

Table 2: Speedup and efficiency for the three platforms

Platform	Speedup	Area	Average Power	Area Efficiency (Speedup/Area)	Energy/Frame (J)
Intel Core i5	1	1	1	1	8.88
Nvidia GTX 8800	3.21	2	2.17	1.61	5.98
CPE (3×3)	10.03	0.04	0.028	245.19	0.025
CPE (4×4)	24.51	0.066	0.0422	369.15	0.015

that cycle). The maximum bandwidth provided by the GPU is calculated from the NVIDIA programming guide and the specification for the 8800 GTX GPU that we used [16], while the internal bandwidths are obtained from code inspection and metrics from the CUDA profiler.

6.2.1 Speedup and efficiency

To compare speedup, we consider the configurations of each platform that come closest the real-time application requirement of 30 fps for all image sizes. Thus, we use the 4-way multithreaded versions of the implementation on the CPU, the 256 threads/block setting on the GPU and the 9 and 16 LE versions of the CPU for our comparison. Table 2 shows the speedup and efficiency numbers for the CPU, GPU and two versions of the CPE (9 and 16 LEs). We directly compare the CPU, implemented in the $32nm$ node, and our other platforms implemented in 90nm technology, as the other platforms are still more efficient. The speedup is calculated for an image size of 512×512 (with 100 iterations), using the execution time on the CPU as the baseline. All numbers except energy consumed have been normalized to the 4-way multithreaded CPU version of SRAD.

The GPU sees speedup because the GPU exploits the data level parallelism in the application with fine-grained threading and efficient memory bandwidth utilization. The two different versions of the CPE shows speedup as expected due to their application specific construction. The speedup numbers seem relatively small for both the GPU and CPE, because the comparison is made to a state of the art CPU at 32nm. The area of the GPU is also twice that of the CPU, which is not a fair comparison as a powerful CPU such as the one we have used would likely occupy much higher area in 90nm. We find the scaled the CPU area at 90nm to be $1920mm^2$ which is 4x larger than the GPU and 120x larger than the CPE. The scaling factors used are obtained from related work in the field [10].

As expected, the disparity in area efficiency (defined as speedup/area) across platforms is bigger than the speedup alone. The CPU sacrifices area for full flexibility and is hence the least efficient, while the GPU is less flexible and utilizes its area for massive number of parallel computational units making it $1.6x$ more area efficient than the CPU although its actual area is twice that of the CPU. The CPE, being highly customized, is the most area efficient of the platforms at the expense of being completely inflexible. The CPE shows up to $153x$ improvement in area efficiency over the GPU. Also, while the GPU consumes $1.5x$ lesser energy for a frame than the CPU, the CPE consumes up to $390x$ lesser energy than the GPU. The energy consumption in the GPU is lesser than the CPU, although it is implemented in much older technology, underscoring the need to move toward specialization in hardware. Since our comparison is with desktop class CPUs and GPUs, we expect that the CPE would show much larger improvement over mobile class processors.

6.2.2 Bandwidth

The memory bandwidth requirements, in Gigabytes/second, for CPE in comparison with other platforms are shown in Table 3. The processed frame is 100 iterations on an image of 512×512 pixels. As shown, the CPU bandwidth utilization is very low for both the L1 and L3 caches because the CPU intersperses memory access instructions with compute instructions, leaving the L1 data cache idle for many cycles before the next memory request. Memory utilization of L3 is even lesser as the locality properties of the application allow good utilization of the L1 and L2. In contrast, the GPU utilizes a large part of the external memory bandwidth by allowing threads to fetch bytes from memory as often as possible, stopping when context storage prevents more threads from being scheduled while other threads wait for their data. We believe that by optimizing threads to work on independent chunks of the image, memory related stalls will be reduced, but may not provide substantial improvement in performance. Shared memory bandwidth is minimal for the GPU as each SP core stores its pixel data in local registers reducing the number of accesses to shared memory. In all, the bandwidth requirement for the CPE is the least among all the platforms since the CPE re-uses fetched data which eliminates redundant accesses by storing intermediate results locally.

To understand the large disparity in bandwidth numbers, we analyzed the memory usage of each platform by analyzing the kernel code and estimating the number of bytes read from memory for processing each pixel. We determined that 76 bytes are accessed independently from global memory for each 4 byte pixel. This is attributed to the fact that, although the code was optimized to share data from the SM's shared memory in a single thread block, pixels on the edge from outside the thread block are fetched redundantly and values intermediate to the two stages of the algorithm are stored back in global memory by each thread.

There are some non-coalesced memory accesses attributable to fetching the west and east pixels from outside the thread block, which cost performance. While it might be possible to reduce the number of redundant fetches, this might also cause threads to diverge. Given that shared memory size was a performance bottleneck, eliminating redundant fetches in the implementation will also reduce the amount of shared memory used per thread. The large amount of memory accessed per pixel translates to the large amount of memory required per frame. We leave the evaluation of further optimizations to future work, but believe that even with a small gain in GPU performance, our comparative analysis will hold. The disadvantage in the GPU is that CUDA code

Table 3: Bandwidth comparison for the three platforms for a 512 x 512 image for 100 iterations

	Internal Memory Bandwidth (GB/s)			External Memory Bandwidth (GB/s)			External Bandwidth per core (GB/s)			Data Fetched per frame (MB)		
	CPU	GPU	CPE (3×3)	CPU	GPU	CPE (3×3)	CPU	GPU	CPE (3×3)	CPU	GPU	CPE (3×3)
Utilization	1.26	1.18	2.75	1.72	66.78	0.098	0.43	4.13	0.01	1200	1900	0.75
Maximum	38.4	47.6	2.75	384	86.4	2.5	96	4.16	0.27	–	–	–

has to be written in a way that does not depend on thread blocks being executed in a particular order, while the CPE implementation relies on row by row execution.

After analyzing the CPU version, we are able to determine that, on average, 76 bytes are accessed independently from the L1 and 3 bytes are accessed independently from the L3 for each pixel since all neighboring pixel data is fetched redundantly. The bytes transferred from the L3 are low due to the good spatial locality in the application. Despite the external memory accessed per frame being lower than the GPU, the CPU does not utilize memory bandwidth effectively to deliver performance. This implies that if more number of computational units were assigned to one cache, similar to the GPU's shared memory structure, there would be better utilization of both the internal and consequently, the external memory bandwidth.

In one iteration, the CPE accesses external memory only once per pixel. This reflects the minimum number of accesses required by the application itself. For each pixel in the frame, the CPE accesses only the pixel value directly, which corresponds to 3 bytes in the CPE representation. The neighboring pixel values are indirectly obtained by the dataflow within the CPE, enabled by the extra storage with the FIFOs. Hence, the CPE external bandwidth requirement is constant for a fixed number of logic elements as the number of operations per cycle is fixed. This information is reflected in 3 as the bytes accessed per frame by the CPE are only 75MB.

7. DISCUSSION

The external bandwidth advantage of the accelerator is attributed to two factors: 1) a portion of the image can be operated upon independently of the remaining portions of the image and 2) the dataflow mechanism which eliminates unnecessary writes to the shared system memory. This makes the design scalable to much higher image sizes as well. However, if an operation on the whole image has to be performed in between iterations, then the architecture will have to be modified to support that operation. In the case of SRAD, if the q0 parameter was not fixed between iterations, some additional logic could alleviate this overhead with very minor cost to throughput. The dataflow mechanism exploited in this accelerator architecture can be extended to support other stencil-based computations as well. Synthetic aperture radar systems could also benefit from this approach.

We believe that an embedded SoC will not be able to deliver the high memory bandwidth the GPU needs to meet performance goals, as the embedded versions of the GPU's use LPDDR and similar technology [2]. LPDDR and LPDDR2 are found to be in use in many of today's high performance embedded systems and provide up to 8.5 GBytes/sec to the system. Existing LPDDR bandwidths are sufficient to support the maximum requirement of 175 MB/s for a 16 LE CPE. Due to this low bandwidth requirement, the CPE will outperform the mobile class GPU as well. Embedded CPUs, much like the desktop CPU, will not be able to meet performance requirements as they are incapable of utilizing DLP effectively. While a (3×3) CPE is capable of meeting performance requirements, we believe that with further optimizations in the CPE pipeline, even a smaller CPE (2×2) can be used, resulting in better efficiency.

We assume the external system resides off-chip, given that this design is targeted at an embedded application. In a portable ultrasound machine for instance, one would find the need for various control mechanisms for operations such as beamforming, filtering, demodulation, log compression and 2D image enhancement. We envision an embedded system that provides quality images by using the SRAD accelerator for image enhancement. The use of this accelerator is orthogonal to use of other solutions for the remaining tasks. As long as the bandwidth requirement of the accelerator is met, either with off-chip DRAM or on-chip SRAM, the system operation should not be affected by addition of this accelerator. As all the operations in the ultrasound system are performed in order, image enhancement will not be the bottleneck as long as the required framerate is met.

8. CONCLUSION AND FUTURE WORK

We have presented the architecture of an accelerator targeted at the Speckle Reducing Anisotropic Diffusion application. The SRAD algorithm is divided into two stages of computation. While other implementations include synchronization between the two stages, our architecture uses a custom dataflow mechanism to make the synchronization implicit. Along with the custom storage for intermediate results, our architecture shows improved performance while also maintaining area efficiency and low energy per task. We also present a comparative analysis of three different platforms for the SRAD application individually and highlight their bottlenecks. Our results suggest that the CPU is compute bound, the GPU is bounded by the memory limitations and CPE scaling is mainly limited by area and storage constraints of the particular system it resides in. Although the GPU implementation is open to slight optimization, even a desktop class GPU is not able to meet the performance requirements of the SRAD algorithm for scaled image sizes. The CPE is capable of meeting this requirement while consuming lesser energy than the GPU. We also find that the system memory bandwidth utilization of the GPU is the highest (and that is where it draws its performance from), that of the CPU is lesser and that of the CPE is the lowest. In typical system on chip implementations, the CPE provides the best advantage as it meets the low bandwidth requirements as well. From our analysis, we conclude that the CPE is the only platform capable of meeting the performance and quality requirement of large image sizes for image enhancement using SRAD. It also remains the most efficient in terms of area and energy consumption for each frame. Future work should optimize the CPE design further by adding more pipeline stages in the data path. Specifically, we have not pipelined any of the functional units such as the divider, multiplier and adder. These optimizations could provide substantial performance improvements. Finally, there are other possible implementations of the SRAD algorithm, such as on DSPs and FPGAs which have different bandwidth requirements, power and area efficiencies; we leave the investigation of these platforms to future work.

9. ACKNOWLEDGMENTS

The authors wish to acknowledge Dr. Nagarajan Kandasamy for lending the GPU cluster for measurements and Steven Battle for his valuable suggestions. We also thank the anonymous reviewers for their excellent feedback. This material is based upon work supported by the National Science Foundation under Grant # CCF-1017654.

10. REFERENCES

[1] J. Agron and D. Andrews. Building heterogeneous reconfigurable systems with a hardware microkernel. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 393–402. ACM, 2009.

[2] Anandtech. Tegra 2 review. http://www.anandtech.com/show/2911.

[3] P. Chaparro, J. González, G. Magklis, Q. Cai, and A. González. Understanding the thermal implications of multi-core architectures. *IEEE Transactions on Parallel and Distributed Systems*, pages 1055–1065, 2007.

[4] S. Che, M. Boyer, J. Meng, D. Tarjan, J. Sheaffer, S. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[5] E. Chung, P. Milder, J. Hoe, and K. Mai. Single-chip heterogeneous computing: Does the future include custom logic, fpgas, and gpgpus? In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 225–236. IEEE Computer Society, 2010.

[6] J. Cong, V. Sarkar, G. Reinman, and A. Bui. Customizable domain-specific computing. *Design & Test of Computers, IEEE*, 28(99):1–1, 2011.

[7] G. Dasika, K. Fan, and S. Mahlke. Power-efficient medical image processing using PUMA. In *Application Specific Processors, 2009. SASP'09. IEEE 7th Symposium on*, pages 29–34. IEEE, 2009.

[8] GPGPU. GPGPU.org. http://gpgpu.org/.

[9] M. Horowitz. Scaling, power and the future of CMOS. In *Proceedings of the 20th International Conference on VLSI Design held jointly with 6th International Conference: Embedded Systems*, page 23. IEEE Computer Society, 2007.

[10] W. Huang, K. Rajamani, M. R. Stan, and K. Skadron. Scaling with design constraints: Predicting the future of big chips. *Micro, IEEE*, 31(4):16–29, july 2011.

[11] Intel Inc. Intel® Xeon® Processor E5620. http://ark.intel.com/Product.aspx?id=47925.

[12] R. Kumar, K. Farkas, N. Jouppi, P. Ranganathan, and D. Tullsen. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 81. IEEE, 2003.

[13] K. Lahiri, S. Dey, D. Panigrahi, and A. Raghunathan. Battery-driven system design: A new frontier in low power design. In *Proceedings of the 2002 Asia and South Pacific Design Automation Conference*, ASP-DAC '02, pages 261–. IEEE, 2002.

[14] Y. Lu, C. Shen, and C. Chen. A novel hardware accelerator architecture for MPEG-2/4 AAC encoder. In *Multimedia and Expo, 2004. ICME'04. 2004 IEEE International Conference on*, volume 2, pages 1139–1142. IEEE, 2004.

[15] A. Mahesri, D. Johnson, N. Crago, and S. Patel. Tradeoffs in designing accelerator architectures for visual computing. In *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, pages 164–175. IEEE Computer Society, 2008.

[16] NVIDIA. CUDA programming guide 2.0. http://developer.download.nvidia.com.

[17] I. Sauciuc, H. Erturk, G. Chrysler, V. Bala, and R. Mahajan. Thermal devices integrated with thermoelectric modules with applications to CPU cooling. ASME, 2005.

[18] N. Sebastiao, T. Dias, N. Roma, and P. Flores. Integrated accelerator architecture for DNA sequences alignment with enhanced traceback phase. In *High Performance Computing and Simulation (HPCS), 2010 International Conference on*, pages 16–23. IEEE, 2010.

[19] P. Shivakumar and N. Jouppi. CACTI 3.0: An integrated cache timing, power, and area model. Technical report, Citeseer, 2001.

[20] D. Terpstra, H. Jagode, H. You, and J. Dongarra. Collecting performance data with PAPI-C. *Tools for High Performance Computing 2009*, pages 157–173.

[21] P. Toonkum, P. Boonvisut, and C. Chinrungrueng. Real-time speckle reduction of ultrasound images based on regularized savitzky-golay filters. In *Bioinformatics and Biomedical Engineering, 2008. ICBBE 2008. The 2nd International Conference on*, pages 2311–2314. IEEE, 2008.

[22] W. Wu, S. Acton, and J. Lack. Real-time processing of ultrasound images with speckle reducing anisotropic diffusion. In *Signals, Systems and Computers, 2006. ACSSC'06. Fortieth Asilomar Conference on*, pages 1458–1464. IEEE, 2006.

[23] C. Xia, A. Zhao, and D. Liu. Optimized GPU framework for ultrasound B-Mode imaging. In *Bioinformatics and Biomedical Engineering (iCBBE), 2010 4th International Conference on*, pages 1–4. IEEE, 2010.

[24] Y. Yu and S. Acton. Speckle reducing anisotropic diffusion. *Image Processing, IEEE Transactions on*, 11(11):1260–1270, 2002.