Mobile Ultrasound Imaging on Heterogeneous Multi-Core Platforms

Andreas Kurth[*], Andreas Tretter[*], Pascal A. Hager[†], Sergio Sanabria[‡], Orçun Göksel[‡], Lothar Thiele[*], Luca Benini[†]

[*]Computer Engineering and Network Laboratory
ETH Zurich, Switzerland

[†]Integrated Systems Laboratory
ETH Zurich, Switzerland

[‡]Computer Vision Laboratory
ETH Zurich, Switzerland

{andkurt, atretter, phager, ssanabria, ogoksel, thiele, benini}@ethz.ch

Abstract—**Ultrasound imaging is one of the most important medical diagnostic methods. The bulkiness of state-of-the-art high-quality ultrasound devices, however, drastically limits their usability in important application scenarios. In this paper, we show how a portable medical ultrasound device can be built using many-core technology and programmable logic, combining low power consumption with high flexibility. We discuss a typical ultrasound image reconstruction algorithm and how it can be parallelized using a pipelined design that efficiently partitions the workload among heterogeneous processing elements. A special focus lies on the limited memory resources and data bandwidth between components. To tackle both problems, we use floating window buffers and approximate computations, and we minimize lookup table sizes using on-the-fly calculations. We evaluate the design on the Adapteva Parallella platform, which contains a power-efficient 16-core Epiphany coprocessor and a Zynq SoC including a dual-core ARM A9 processor and programmable logic. Experimental results show that parallel beamforming of 128 input channels to a 288x128 pixel ultrasound image can be achieved on the Parallella at a rate of 5.3 frames per second consuming only 2 watt of dynamic power.**

I. Introduction

One of the most important medical diagnostic methods is ultrasound imaging. It is non-invasive and free of possibly harmful radiation and allows, at low cost, an instant-feedback examination performed directly by the practitioner. Unfortunately, state-of-the-art high-quality ultrasound devices – as used in hospitals or doctor's offices – are at least comparable to desktop PCs in size and have a significantly higher power consumption. This limits their action radius to the room they are located in, whereas it would often be desirable to have them available on site, e.g., in emergency situations, in rural areas (particularly so in developing countries), or for military applications.

This bulkiness does not come from ultrasound imaging itself – the function principle is simple and ultrasound probes are small and handy. The problem is the high amount of data (gigabytes per second) that needs to be processed in real-time, which means high computational costs on top of a high data exchange bandwidth. Meeting these demanding requirements on a mobile low-power platform can only work with a sophisticated design that is carefully optimized on many different levels, such as algorithm, hardware components, task parallelization, scheduling, and data exchange.

How can such a design look like if the goal is to build a mobile ultrasound device that needs to run on battery power for several hours? And if, according to the utilization scenario for developing countries, this device should be financially affordable and flexible enough to support many different imaging methods and algorithms?

The last decades have seen ultrasound systems based on many different hardware platforms, such as ASICs, FPGAs, multi-core DSPs, or even high-end GPUs. The recent trend shows many innovations especially in the area of software-based ultrasound imaging on commodity hardware, which not only reduces cost, but also gives access to cutting-edge technologies that would not be affordable in custom designs produced at low quantities. Meeting power saving demands, however, is difficult with these solutions.

In the past years, new interesting hardware architectures for embedded computing have emerged: Homogeneous and heterogeneous multi- and many-core systems provide high compute efficiency and have been extended with co-integrated programmable logic to provide in-system configurable hardware acceleration. Yet, until now, it has not been discussed how both many-core technology and programmable logic can be used simultaneously for accelerating computations.

In this paper, we evaluate the opportunities of combining multi-core with programmable logic accelerators. The focus will be on future mobile ultrasound systems, but most of the ideas and techniques we present are also applicable or can be generalized to other application domains.

We will discuss the challenges that lie in programming these systems and in taking advantage of their heterogeneity. For instance, distributing tasks among the different components is not only a question of efficient execution on the target resource type, but also of dependencies between the tasks, of task synchronization, and of data transport. Other problems that need to be tackled are the memory limitations of embedded multi-core systems, the limited bandwidth on the connection between the different components, and execution sequentialization due to inappropriate task synchronization.

We will describe a toolset of three methods that address these problems. This involves how an ultrasound imaging algorithm can be parallelized and distributed between multiple RISC cores and programmable logic in a pipelined fashion, such that parallelism can be optimally exploited at a minimal synchronization overhead. It involves different methods for reducing memory and bandwidth requirements, which actually make this mapping possible, such as online and on-demand calculation of coefficients. Furthermore, it involves approximation during image reconstruction to achieve significant speed-ups at the cost of only minimal quality deterioration.

Finally, as a proof of concept, we will show an implementation of our design on the Adapteva Parallella platform, which features both a Xilinx Zynq system-on-chip (SoC), combining an ARM host processor with programmable logic, and an energy-efficient 16-core coprocessor.

This paper is structured as follows: Section III presents the function principle and implementation details of ultrasound imaging. Section IV shows our target architecture model. Section V fully discusses the challenges that lie in ultrasound image processing on that architecture. In Section VI, we present the methods we applied to solve those problems. We show the concrete results on the Parallella platform in Section VII, together with calculations on how to scale this system to a real-time implementation. To start, the next section compares our contribution to existing work.

II. Related Work

Full-fledged high-end ultrasound systems are generally built using multiple field-programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs): One of the most powerful and flexible ultrasound systems is the SARUS [1], [2] research system, which is designed for maximal flexibility. It features 320 Xilinx Virtex-4 FPGAs and more than 320 GB of memory to implement most advanced imaging algorithms in real time using data from 256 fully independent receivers. This system uses FPGAs for maximum flexibility and processing power. Low costs, power efficiency, or high integration are not the target. Even when flexibility is not required, multiple FPGAs are necessary to provide sufficient compute power: In [3], an ultrasound system using 4 Xilinx Virtex-5 FPGAs is presented. While ASICs would provide the same compute power, FPGAs are often chosen since they are simply cheaper for the low product volumes in which ultrasound systems are generally sold. When size matters and maximal power efficiency is required, ASICs are used. With a clear focus on miniaturization and power-efficiency, [4] presents a fully-integrated ultrasound image processor supporting real-time imaging of 100 receivers, while consuming only 303 mW at a silicon area of 1.675 mm^2 in an advanced 22 nm technology. Even though this system provides some reconfigurability and superior power efficiency, it relies on single-purpose hardware with complex economical dependencies on market volume and pricing.

Recently, a shift towards more software-based processing can be observed in ultrasound systems: In [5], ultrasound image processing of 64 receivers is implemented using a 8-core TI TMS320C6678 KeyStone digital signal processor (DSP). In both commercial [6] and research [7] systems, graphics processing units (GPUs) are used for processing. The latter uses two NVIDIA GTX 480 to support 32 receivers. Regarding many-core processors, [8] demonstrates ultrasound imaging for 63 receivers on the Intel Xeon Phi 5110P with 60 cores.

This shift to software processing is partially owed to the advent of powerful and efficient parallel processing architectures including large-scale streaming processors and multi- and many-core systems [9]–[12]. The other, equally important reasons are recent innovations on the software side providing programming models [13]–[15] that greatly simplify the programming of these complex systems. As such general-purpose processing devices can be used in many applications, they profit from economy of scale and are thus relatively cheap, which makes them ideal for our intended application. Nevertheless, current software implementations are based on platforms that are not suitable for mobile devices.

Our implementation differs from the aforementioned in two ways: First, we use a *combination* of programmable logic and a multi-core coprocessor. Second, we leverage the heterogeneity of the platform to bring a software-based implementation to the domain of low-power mobile devices.

As one of the two central components, we use the Epiphany [12] low-power multi-core processor. [16] gives an in-depth overview of Epiphany and discusses two different benchmarks to show how to effectively optimize code for that processor. That work focusses on execution optimization within the Epiphany and discusses only some general considerations on how to use it in a bigger system. In contrast to that work, we integrate the Epiphany in a processing system, discuss the challenges that arise and present methods to tackle them, and prove the concept with results on the overall performance of the processing system.

In the area of computation acceleration, [17] and [18] describe how to offload computations to programmable logic, and [19] describes offloading to a multi-core coprocessor.

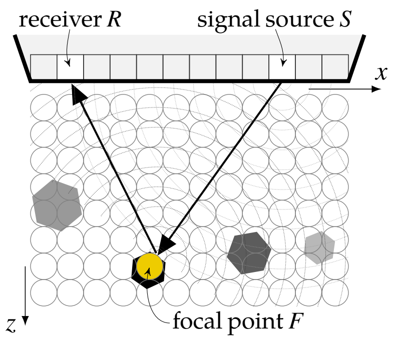


Figure 1: Geometric setup of synthetic aperture imaging.

However, to the best of our knowledge, there is no previous work on how to use both types of accelerators *simultaneously* for application speed-up.

III. Ultrasound Imaging

This section gives a brief introduction to medical ultrasound imaging. First, the general principle of operation is explained, followed by the imaging method. Then, after a system overview, the implemented image processing algorithm is described in detail.

A. Principle of Operation

Ultrasound imaging is based on the principle of acoustic reflection. A sound wave incident on an acoustic impedance boundary is partially reflected back to the emitter. Piezoelectric crystals called *transducers* are used to emit sound waves as well as to record the reflections. A single such emit-receive cycle is called a *shot*. Assuming a constant wave propagation speed in the tissue, the distance between transducer and object boundary depends linearly on the arrival time of the reflected wave. A simple reflectivity chart can thus be constructed by plotting the amplitude of the received signal over time (*amplitude* or *A mode*).

By performing multiple such shots while sweeping the transducer sideways, one obtains an array of *scanlines*. It can be displayed as a gray-scale image by representing the echo amplitude as the brightness of a pixel (*brightness* or *B mode*). This image type is the most widespread in medical ultrasound. Modern ultrasound probes, however, do not mechanically sweep a single transducer; instead, they send different signals to an array of fixed transducers in order to emulate the same effect.

A drawback of B mode scanning is that many shots are required to compose one image. Emitting a sound wave comes at considerable energy costs. In mobile applications, the number of emissions should thus be minimized. In order to do so, we use an imaging method technique called *synthetic aperture*.

B. Synthetic Aperture Imaging

In *synthetic aperture imaging* [20], an unfocused wave is emitted into the tissue and the entire image is reconstructed at once from the recorded echo traces. As one emission yields only poor image quality, the reconstructed images from multiple shots using different unfocused waves are combined.

The unfocused emission is generated by exciting a single transducer, which we call the *signal source* S (Fig. 1). Let us now consider one particular point in the tissue, which we call a *focal point* F. The wave travels from S to F. If there is a change of acoustic impedance at F, the wave is scattered and the reflection travels back to the transducers, which now act as *receivers*. Let us pick one of them and denote it as R. When the reflection arrives at R, the distance covered by the outgoing and the reflected wave is $\overline{SF}+\overline{FR}$. Following the assumption of constant wave propagation speed, the time at which the

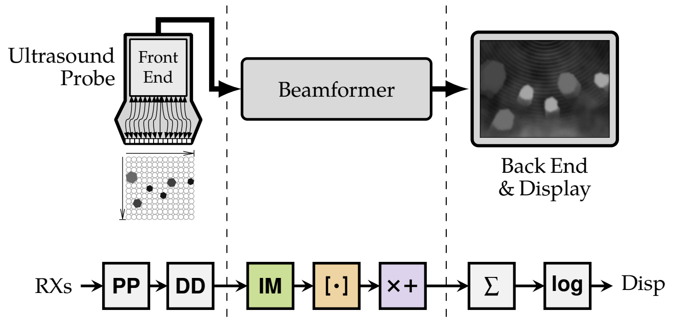


Figure 2: Components (top) and signal processing steps (bottom) in a parallel beamforming system.

reflection arrives at R can be computed and the corresponding sample index k in the trace of R can be determined:

$$k_{S,R}(F) = \left\lfloor \left(\overline{SF} + \overline{FR}\right) \cdot \frac{f_s}{v_0} \right\rceil, \tag{1}$$

where v_0 is the sound speed in the tissue and f_s the sampling frequency of the traces. $\lfloor \cdot \rceil$ denotes rounding to the nearest integer.

If a reflection occurred at F, the signal amplitude will be high at sample k in the trace of R. Yet, if no reflection occurred at F, the amplitude could still be high due to a reflection at some other point. This ambiguity, however, can be eliminated by superposing the contributions of all receivers. The presence of a scatterer at F leads to constructive interference of the echoes; the absence of a scatterer leads to destructive interference of noise. Performing this calculation for each point in a grid of focal points finally yields the reconstructed image. We refer to this entire process as *beamforming*.

As one such image may still contain noise, the reconstructed images from multiple shots are combined (i.e., averaged) into one *frame*. The different unfocused waves for these shots are generated by selecting a different transducer as source for each shot. We assume that each transducer can act as a signal source.

C. Technical specifications and system overview

This section details the specific imaging system that we implement in this work. It is illustrated in Fig. 2 and comprises three major parts: First, a probe (with an integrated front end) that emits acoustic pulses and captures the echo traces. Second, a beamformer that processes each shot as described previously, thereby creating *reflectivity maps*. Third, a back end that combines the reflectivity maps from multiple shots and transforms them into a grayscale image targeted at human perception.

In this work, a probe with $N = 128$ transducers is used. Each transducer output is processed into a digital trace of $K = 3080$ complex-valued samples per shot.

To save transmission bandwidth between the probe and the ultrasound device, the traces are demodulated and decimated (DD) with a factor of $Q = 8$. This process is reversed on the beamformer by a corresponding interpolation and modulation (IM) step. In the following, we refer to these operations as *compression* and *decompression*.

In the remaining two steps of the beamformer, the traces are transformed into still complex-valued reflectivity maps that consist of 128 scanlines with 288 focal points each. For each focal point, sample selection ($[\cdot]$) finds the corresponding samples according to (1) and apodization and accumulation ($\times+$) combine the contributions of multiple receivers.

Finally, the back end sums up the reflectivity maps of 4 shots from different transducers into one frame. This frame is then converted into a grayscale image in which the brightness of each pixel is computed from the logarithm of the absolute value of the corresponding complex frame value. This simple post-processing is sufficient for basic B mode imaging and requires little processing effort. To avoid motion blur and to achieve a smooth display, we assume that a frame rate of 20 frames, i.e. 80 shots, per second is necessary.

Relative to the whole imaging system, most of the signal processing workload is incurred in the beamformer. Therefore, we will concentrate on that part in the rest of this paper.

D. Beamforming Algorithm

This section details the beamformer part. As discussed before, the individual computation steps for one shot consist of the beamforming process itself and the preceding interpolation and modulation. They are shown in Fig. 3.

In the interpolation and modulation step, the incoming 128 compressed traces r_n^{compr} are represented as complex values. First, they are decompressed, i.e., interpolated and modulated, to 3080 samples per trace: The interpolated traces r_n^{interp} are obtained from the r_n^{compr} by *zero-stuffing* (i.e., adding $(Q-1)$ zero samples between each compressed sample) and convolving with a low-pass filter of order 6. Then, r_n^{interp} is modulated to restore the original traces r_n by multiplying each sample with a complex unity factor:

$$r_n[k] = \exp\left(2\pi \mathrm{i} \cdot \tfrac{k}{Q}\right) \cdot r_n^{\text{interp}}[k], \quad 1 \le k \le K. \tag{2}$$

The next step is sample selection. In this step, for each focal point F in a grid of dimension 288×128, the corresponding sample is retrieved from each trace according to (1). We use the coordinate system shown in Fig. 1: The x-axis runs parallel to the transducers and the z-axis goes into the tissue. To simplify the calculations, we set the unit length for the axes to $\frac{v_0}{f_s}$. With the location $S(x_S, 0)$ of the signal source and of the n-th receiver $R_n(x_{R_n}, 0)$ given, the sample for each focal point $F(x,z)$ is

$$p_n(F) = r_n\left[\left\lfloor \sqrt{(x-x_S)^2 + z^2} + \sqrt{(x-x_R)^2 + z^2} \right\rceil\right], \; 1 \le n \le N. \tag{3}$$

We refer to the different $p_n(F)$ as *partial* values, since all these values are later needed to calculate the reflectivity value for F.

In the last step, the reflectivity map I is formed by summing up the partial shot data of all receivers. To suppress side lobes caused by the aperture shape [4], the different partial values need to be weighted:

$$I(F) = \sum_{n=1}^{N} w_n(F) \cdot p_n(F), \tag{4}$$

where w_n are the weighting factors. This weighting is called *apodization*.

The partial values in (3) are independent of each other and can be computed in parallel. This makes beamforming feasible and attractive for implementation on parallel processing platforms.

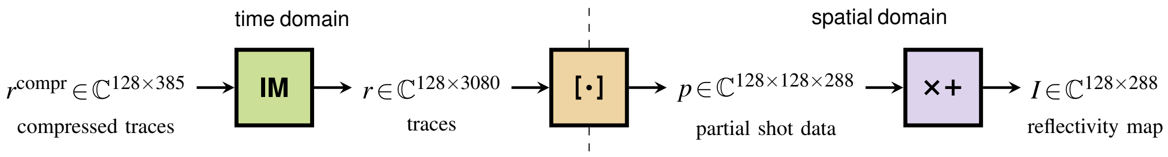

time domain
spatial domain
$r^{\text{compr}} \in \mathbb{C}^{128\times385}$
compressed traces
IM
$r \in \mathbb{C}^{128\times3080}$
traces
[·]
$p \in \mathbb{C}^{128\times128\times288}$
partial shot data
×+
$I \in \mathbb{C}^{128\times288}$
reflectivity map

Figure 3: Overview of the individual beamforming steps and results.

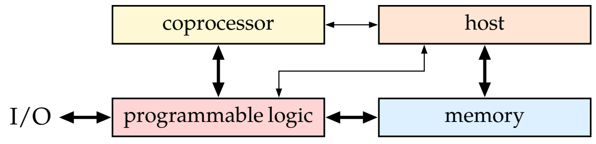


Figure 4: General structure of the considered target architecture. Thick lines represent high-bandwidth connections intended for data transport. Thin lines represent low-bandwidth connections.

IV. Heterogeneous Multi-Core Platform

This section gives details about the considered target architecture for the algorithm that was previously introduced. After a brief overview of the components and the general structure, the EPIPHANY E16G301 is presented as a prototype for an energy-efficient multi-core coprocessor. Finally, as an example for a possible target platform, the PARALLELLA is shown.

A. System Overview

Figure 4 outlines the general architecture of the target platforms we consider. It contains a host processor that runs an operating system and manages all resources and interfaces. The host does not participate in handling the ultrasound workload and therefore is not required to scale with the problem size. This is particularly important with respect to power consumption: The host processor is typically not optimized purely for energy efficiency, but instead should be both sufficiently powerful to accommodate an operating system and flexible to support various interfaces to connect, e.g., network links, storage devices, or displays, depending on the application needs.

A multi-core coprocessor and a programmable logic (PL) component perform the computationally intensive tasks. The different tasks are divided among both components; as a rule of thumb, tasks with regular control flow are implemented as custom logic in the PL while those with irregular control flow are executed on (some cores of) the coprocessor.

Input streams, e.g., from the probe, can be directly connected to the PL via input/output (I/O) interfaces. High-bandwidth connections for efficient data transfer are established between PL and coprocessor as well as between PL and memory. The host controls PL and coprocessor through low-bandwidth connections as data transfers happen via shared memory. Coprocessors usually do not include a memory controller to maximize silicon area for processing elements; instead, they access the system memory through the PL.

B. EPIPHANY *Multi-Core Coprocessor*

In the rest of this paper, we will use the EPIPHANY [12] by ADAPTEVA as a concrete example of the multi-core coprocessor in the abstract architecture described above. Specifically, we will use the E16G301 model featuring 16 cores. The EPIPHANY architecture suits our low-power requirements since it was designed with a clear focus on energy efficiency (i.e., maximum floating-point operations per second (FLOPS) per Watt). Additionally, the architecture was designed for scalability of compute capacity with problem size through scaling the number of cores.

The EPIPHANY E16G301 chip contains 16 cores arranged in a 4×4 grid. Each core is equipped with an ALU, a load-store unit, and a floating-point unit (FPU) that can work in parallel to the other two units in a superscalar fashion. As cores are optimized for area and energy efficiency, only a small set of instructions corresponding to the typical signal processing operations are supported. 32 KiB of local on-chip memory are available per core, together with a DMA controller. The memory of each core is divided into 4 banks, allowing multiple simultaneous accesses like instruction fetching, data load/store, and DMA operations.

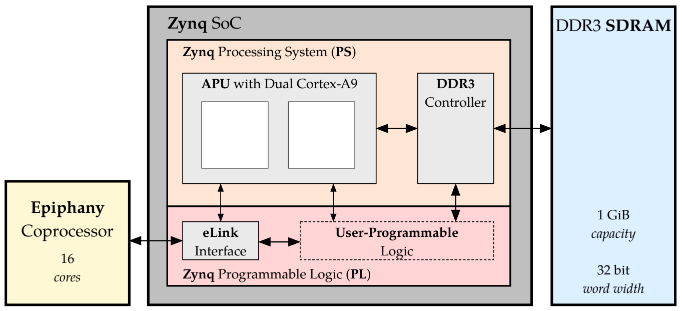


Figure 5: Simplified high-level architecture overview of PARALLELLA.

Active Component A	Passive Component P	Connection Bandwidth $A \rightarrow P$	Connection Bandwidth $P \rightarrow A$
Host	Memory	92.4 MiB/s	138.3 MiB/s
PL	Memory	284.2 MiB/s	274.9 MiB/s
Coprocessor	PL	121.1 MiB/s	88.6 MiB/s

Table I: Effective bandwidth for different connections on the PARALLELLA *platform.*

The cores are connected through a network-on-chip (NoC) arranged in a two-dimensional mesh topology with only nearest-neighbor direct connections. 64-bit packets are routed through the NoC at one hop per cycle. Communication takes place via memory accesses: All memory banks are part of one single address space and each core can access the memory of any other core. Store operations on remote memory banks are implemented as packet transmissions without flow control. As such, they take only one cycle to execute but may have a much longer latency. Therefore, a core can continue its computations after a store operation, while the data is travelling to the destination memory block. As a result, streaming applications without cyclic data dependencies can be executed efficiently by storing calculation results directly in the memory of the core that requires the results.

C. Evaluation Platform

We use PARALLELLA P1602, a heterogeneous open-source hardware platform by ADAPTEVA, to evaluate our project. Its architecture is illustrated in Fig. 5. The platform corresponds to our model from Section IV-A: Both the host processor and the PL are contained in a XILINX ZYNQ-7020 SoC, the former as a dual-core ARM CORTEX-A9, the latter as 85000 logic cells, 4.9 Mbit of block RAM (BRAM), and 220 DSP slices. An EPIPHANY E16G301, as described above, is available as a coprocessor. The ARM host is clocked at 667 MHz, the EPIPHANY chip is clocked at 600 MHz.

The ARM host and the PL share a DDR3 controller that provides access to 1 GiB of external SDRAM. The EPIPHANY coprocessor is connected to the ZYNQ through an interface implemented in the PL of the ZYNQ. The maximum effective data rates of the platform, which we have determined empirically, are listed in Table I.

V. Challenges

This paper discusses design principles and methods to implement the beamforming algorithm from Section III on the type of platform described in Section IV. This task is far from trivial and has many challenges and difficulties to overcome, as we will show in the following.

The first question we address is: How can the application be mapped to the available computation resources? There are two classic, antipodal approaches: exploiting data parallelism or pipelining.

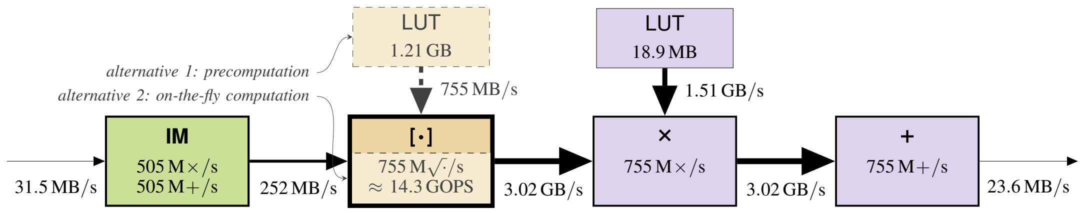


Figure 6: Computational characteristics of beamforming. The numerical values correspond to the imaging parameters introduced in Section III. Task box border and communication arrow weights scale logarithmically with computational complexity and data rate, respectively.

The pure data parallelism approach uses the fact that each output pixel can be calculated independently. In particular, this also holds for the partial values for each pixel (which are afterwards apodized and accumulated). Each computational resource could therefore calculate a distinct set of output pixels (e.g., a row or a column). This approach is typically adopted in GPU programming or hardware design. Such a solution does not profit from the different, complementing types of computational resources.

In the pipelining approach, the algorithm is divided into steps and each step is executed on a different computational element (i.e., PL block or single core). This allows to respect *task affinity*, i.e., the concept of mapping tasks with regular control flow, like interpolation, modulation, or apodization, to the PL (which can efficiently handle this kind of operations) and the less regular tasks, such as delay computation and sample selection, to the multi-core coprocessor. This is the approach taken by software frameworks targeting streaming applications, like STREAMIT [21], DAL [22], or MAPS [23]. Apart from the fact that pipelining alone cannot create enough parallelism, it also enforces inter-core and cross-chip data exchange even where the data rate is highest. Moreover, as the tasks have different execution times and cannot be partitioned arbitrarily, load balancing is hampered and thus the resources are not used optimally.

The second question we address is: How to cope with the limited memory capacities and the limited communication bandwidth? Figure 6 illustrates this problem by means of the beamforming algorithm and the imaging parameters introduced in Section III. As we will pinpoint in this section, sample selection and apodization and accumulation, as well as their interaction, seriously challenge the available memory and bandwidth.

In sample selection, one sample is selected for each of the 128×288 image points of the partial reflectivity map for each of the 128 receivers; i.e., 4.72×10^6 sample indices are used in every shot. Assuming 80 shots per second, 378×10^6 sample indices are used per second. There are two possibilities for obtaining these indices: On the one hand, each index could be calculated when required. The computation of one index involves the calculation of two distances, i.e., two square root operations. As a result, 755×10^6 square roots would have to be calculated per second. Given that hardware support for square root calculation is available neither on the EPIPHANY nor on the PL, it is not clear how to the required computation performance can be achieved. On the other hand, all indices could be pre-calculated and each index could be loaded when required. Pre-computing and storing the indices would take 1.21 GB of memory in total, assuming 128 different signal sources and 16-bit integers for storing the indices. Furthermore, fetching the indices for 80 shots per second would contribute a data rate of 755 MB/s. This would not only hit the limitations of available memory size and bandwidth, but also lead to a high power consumption.

Apodization and accumulation (henceforth treated as two separate tasks to increase mapping flexibility) jointly calculate the weighted sum of indexed samples in (4): every complex-valued partial value is multiplied with a real-valued weight and the results of all receivers are accumulated. Thus, these two tasks involve two floating-point multiplications and additions per partial value, totaling to 9.44×10^6 or 755×10^6 multiplications and additions per shot or per second, respectively. As one weight is unique to image point and receiver (cf. the indices of w in (4)), a table holding all weights occupies 18.9 MB of memory. In every shot, the data from the entire table is read, which amounts to a data rate of 1.51 GB/s.

The high number of computations and the memory size and bandwidth requirements of both the sample selection task individually and the apodization and accumulation tasks together strongly suggest to map the former task to a different component than the latter two tasks. However, partial shot data (cf. Fig. 3) has to be transferred from the former task to the latter two tasks at a rate of 3.02 GB/s. Such a high data rate would clearly hit a bottleneck if this data was to be exchanged between coprocessor and PL.

In summary, implementing the beamforming algorithm on a heterogeneous platform with low-power commodity components poses two general challenges: First, due to the limited computation and memory resources of the individual components, the implementation must exploit all available resources simultaneously by efficiently distributing the workload among all components. Second, the hence required cooperative solution challenges the capabilities of heterogeneous platforms due to the limited available communication bandwidth and the necessity to optimally use the available heterogeneous computing components. Smart trade-offs alone do not solve these problems; instead, more sophisticated methods are required.

VI. METHODS

A single design technique is not sufficient to tackle the diverse challenges shown in the last section. In this section, we explain three methods which we apply and which together lead to a feasible solution. First, we show a *heterogeneous parallel pipeline* – a data processing pipeline whose different stages, which internally are parallel, are mapped to different heterogeneous processing elements – as our approach to the mapping problem. Then, we present how we reduce memory requirements to deal with the fact that EPIPHANY and PL provide very little on-chip memory compared to the problem size. Finally, we introduce two forms of *approximate computing* to decrease computational complexity and to reduce inter-chip data exchanges.

A. Heterogeneous Parallel Pipelining

As discussed previously, mapping the tasks to the different computation resources is non-trivial: even when neglecting constraints like bandwidth or local memory for a moment, we still have to strike a balance between task affinity (i.e., the concept of mapping a task to a resource that has been designed to execute it efficiently) and potential parallelization.

Methodologically, we begin on the coarse-grained level, i.e., on the level of the different resource types. For reasons of efficiency, the top-level design structure should allow us to use the heterogeneous resources concurrently. The two ways to achieve this are functional parallelism and pipelining. The beamforming algorithm does not feature functional parallelism; consequently,

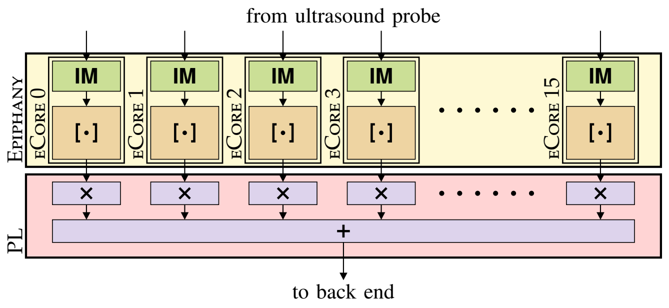


Figure 7: Heterogeneous parallel pipeline mapping of beamforming to the target platform.

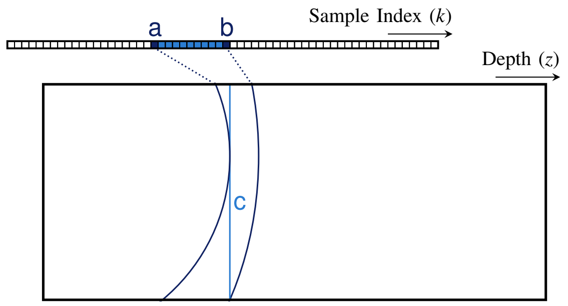


Figure 8: Correspondence between trace samples and pixels in a partial reflectivity map.

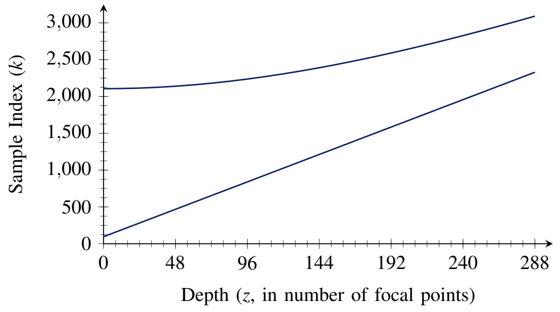


Figure 9: Lower and upper bounds on sample index as a function of imaging depth for different block sizes.

the top-level structure of the mapping is a pipeline. In our case, we decide to map the sample selection (3) to the coprocessor since the data flow of this task is less regular, thus favoring a processor core over PL. The modulation and interpolation step (2) is then also mapped to the coprocessor, since this reduces the input bandwidth of the coprocessor by the interpolation factor. The two remaining tasks, apodization and accumulation (4), feature a highly regular data flow. Thus, they can be implemented very efficiently on PL. Moreover, apodization is the task with the highest data rate, and the connection between PL and memory has a high effective bandwidth (cf. Table I). As a result, both apodization and accumulation are mapped to PL.

Within the coprocessor, many parallel homogeneous processing elements are available. Two options for distributing the workload on parallel processors are pipelining and data parallelism. In order to exploit the high number of parallel data paths in the beamforming algorithm, as well as to avoid synchronization and load balancing issues among cores, we choose data parallelism within the coprocessor. In particular, parallel processing of the different traces offers the highest independence between the cores.

These considerations lead to the heterogeneous parallel pipeline presented in Fig. 7: Each coprocessor core interpolates and modulates one transducer trace at a time. It then selects all the samples that are needed from the interpolated trace, resulting in one partial reflectivity map. All these partial reflectivity maps are then sent to the PL, where they are apodized and accumulated. The 128 transducers are split into 8 groups of 16 transducers each and are, within a shot, processed sequentially by the 16 coprocessor cores.

B. *Buffering, Data Transactions, and Synchronization*

To operate the heterogeneous parallel pipeline at its full capacity with minimal stalls, efficient synchronization and buffering between stages is vital. Our implementation rests on the following three pillars: double buffering, burst transactions executed in background by DMA engines, and fine-grained producer-consumer synchronization with locally-mirrored synchronization variables.

1) Double Buffering: We use double buffering at the input and output of pipeline stages to enable the concurrent operation of all stages. In every pipeline cycle, one buffer at the input is used to fetch the operands for calculations; the results of the calculations are then stored in one buffer at the output. The second input and output buffers are used by DMA engines to transfer data in and out of the stage.

2) Burst Transactions: We transfer data in bursts with DMA engines between different chips for three reasons: First, it allows processing resources to fetch operands from local memory and write results also to local memory. This avoids load/store stalls due to bus congestion. Second, transactions handled by DMA engines leave more clock cycles for processing elements to actually perform data calculations. Third, burst transactions maximize bus utilization during a transaction and minimize total transaction time.

3) Synchronization: We employ producer-consumer synchronization to govern access to memory shared by two pipeline stages. For every shared memory section, there are two synchronization counters: one for the number of bytes written by the producer and one for the number of bytes read by the consumer. The difference between these counters yields the number of bytes that are available for read and, subtracted from the total buffer capacity, the free space in the buffer. Note that this mechanism still works if one or both of the counters overflow. These synchronization variables are mirrored at both parties; they are read from local memory and written to both local and remote memory to exploit the properties of data transfers on the platform. In this way, polling across different computation resources or elements is avoided.

Data exchanges between different cores of the coprocessor work in a similar way; however, no burst DMA transfers are used. Instead, the buffer is located on the destination core and the source core writes data items directly into this buffer.

In summary, we transfer data in bursts between stages and employ fine-grained synchronization with locally mirrored variables to minimize sequentialization stalls at a low communication and memory overhead. Therefore, as synchronization between stages comes at minute costs, we can afford to synchronize often. This, in turn, reduces the size of buffers between pipeline stages.

C. *Minimizing Memory Size and Inter-Chip Data Exchange*

On-chip memory of the target platform is very limited; in particular, each coprocessor core only has 32 KiB of memory. Therefore, we have to find techniques that reduce the amount of memory required for buffers as far as possible.

We first address the buffer that holds the decompressed traces. To minimize its size, we have to understand the correspondence between the samples in the traces and the image points created from them. This relationship is illustrated in Fig. 8: One sample (like a or b in the figure) is required for a particular set of focal points in a partial reflectivity map. This is the set of points for which the distance in (1) from the sending transducer over the focal point to the receiving transducer is constant and corresponds to the time index of the sample. These sets describe ellipses, as can be seen in the figure. The row c of points in the reflexivity map lies between the ellipses corresponding to a or b, which implies that only the samples between a and b are needed for calculating all partial values on this row.

Figure 9 shows the upper and lower bounds for each row (over all possible emitter/receiver combinations) of the sample indices required to calculate it. We take advantage of this information by calculating the partial reflectivity maps row by row, reducing the number of samples that must be available *simultaneously*. These samples are interpolated and modulated as needed and stored in a floating window buffer. In this way, we do not have to allocate a buffer for the entire decompressed trace, but only for the largest part of the trace that is needed for one row. As we can afford to synchronize often, we choose to calculate and transmit one row at a time, thereby minimizing not only the size of the buffer holding the decompressed samples (ca. 15.9 kB), but also the size of the buffers between stages (1 KiB each).

The second important consideration in terms of buffer size concerns whether or not to store the required indices for sample selection. As explained in the previous section, we could avoid 9.44×10^6 distance calculations per shot at the cost of 1.21 GB of memory. As this is not feasible on the target platform, we instead minimize memory requirements and inter-chip data exchanges by calculating indices within each core on demand. With this decision, we have made distance calculations a very frequent operation. Our decision to make use of row-wise processing allows us – through geometrical considerations on the regular grid of focal points (Fig. 1) – to compute the argument of each square root in the sample index calculation (3) with only one multiplication-accumulation and one addition. We tackle the computational complexity of the square root itself with the approximation shown below.

In summary, we use floating window buffers and on-core on-demand calculations to minimize both local and global memory size and reduce inter-chip data exchange. While the latter comes at the cost of added computations, the former does not change the computational complexity at all.

D. Approximate Computing

So far, our implementation exactly computes the algorithm described in Section III. To further increase execution speed and energy efficiency, we apply two forms of approximate computing: we approximate the square root (3) with a *piece-wise linear function* to reduce the complexity of distance calculation, and we approximate the apodization in (4) by *group-wise pre-accumulation* before apodization to reduce inter-chip data exchanges. The incurring losses on image quality are negligible, as we will show in Section VII.

The FPUs in the EPIPHANY cores, which compute the square roots for sample index calculation (3), do not provide hardware support for square root computations. Thus, square roots must be computed in software. Now, the standard C library implementation for the EPIPHANY architecture uses an iterative algorithm (described in [24]) that is computationally intensive. Furthermore, that implementation calculates the square root to full floating-point precision, which we do not require: as long as the result of sample index calculation – which rounds the sum of two square roots to an integer – is precise to within one

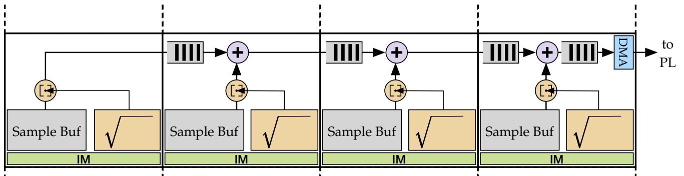


Figure 10: Four ECOREs in one row on EPIPHANY form a group. Over the entire 4×4 grid, four such groups operate in parallel.

sample [25], we can tolerate an approximation loss. Instead of approximating the square root iteratively, we use a piece-wise linear approximation [26] to exploit the fact that both the required domain of the function and the required precision of the result are bound and independent of data, thus known at design time. We use 39 sections over a domain from minimum to maximum squared distance in the sample index space (numerically: $62.5 \times 10^{-3} \,..\, 2.34 \times 10^6$) to keep the error in the codomain strictly smaller than 0.25 (i.e., precise to within one sample after the sum of two approximated square roots is rounded to the next integer). We compute the section index from the argument of the square root with a binary search tree.

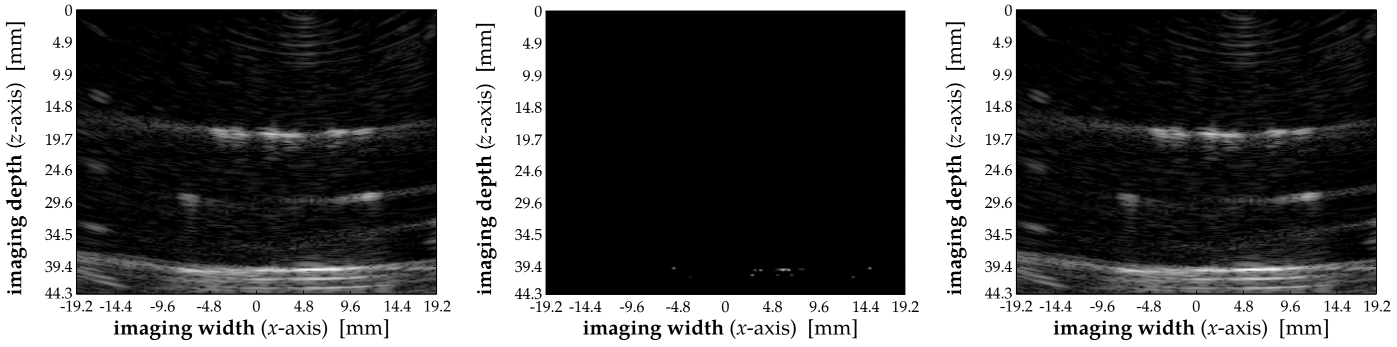
In the mapping presented in Fig. 7, partial values are transferred between chips from sample selection on EPIPHANY to apodization on PL. Recalling Fig. 6, this is also the point where the highest data rates in the beamformer occur. However, as pointed out in Section IV, the capacity of this interface is limited. To avoid this data exchange from becoming a bottleneck, we introduce the *group-wise apodization* approximation. The idea is to, instead of apodizing each partial value with its individual weight, use the same apodization weights for the partial values of neighboring receivers. We combine the receivers into groups of four. For each such group, we calculate the common apodization weights by averaging over the original weights of its four receivers.

This allows us to significantly reduce the data rate at the output of EPIPHANY: Using group apodization, a part of the accumulation can already be performed directly after sample selection on the EPIPHANY without the need of transmitting the corresponding apodization weights. On the PL, these accumulated partial values are then apodized together. Due to the regular 4×4 structure in which cores are arranged, we form groups of four cores as shown in Fig. 10. In contrast to the original mapping, the three leftmost cores in the group forward their partial values to the next core instead of the PL, and the three rightmost cores each add the partial value of the previous core to their own before forwarding the result. The last core in the group finally stores the group-wise pre-accumulated partial values in a local buffer. From that local buffer, the group-wise partial values are transferred using DMA to the PL, where they are apodized.

This modification of the mapping has the advantage that both the output data rate from EPIPHANY and the computational load on the PL are reduced by a factor of four. While the latter is a side-effect, the former is crucial to prevent the data rate from the EPIPHANY from becoming a bottleneck, as we show in Section VII. In the same section, we show that the image quality losses introduced by this approximation are negligible.

VII. RESULTS

In this section, we present the measurements that we performed on our implementation of the heterogeneous parallel pipeline mapping and discuss the results. First, we describe the tools with relevant configuration options. Second, we verify that our approximation and implementation loss is imperceptible by comparing output images to those of a reference implementation. Third, we examine processing performance



(a) Result of reference implementation. *(b) Absolute value of normalized squared error.* *(c) Result of our implementation.*

Figure 11: Visual comparison of the reflectivity map from our implementation to a reference implementation. The images are log-compressed and have a dynamic range of 60 dB. *High absolute values are represented by bright pixels.*

and power consumption to ascertain that the heterogeneous parallel pipeline achieves the aspired energy efficiency. Fourth, we inspect resource usage and execution time to make sure that the overhead of synchronization and double buffering is as low as intended and that the heterogeneous parallel pipeline operates at its full capacity. Finally, we conclude this section by extrapolating our implementation to real-time imaging.

A. Evaluation Setup

We evaluate our implementation on a PARALLELLA P1602. We use the EPIPHANY toolchain version 2015.1 [27], including ELIB, a library providing low-level APIs for programming EPIPHANY, and an implementation of the ELINK (the interface to connect EPIPHANY) on the PL. We have modified ELIB to support non-blocking DMA transfers, DMA interrupts, copies on both DMA engines, and access to memory regions outside the default shared memory regions. On the host processor, we run a LINUX 3.14.12 kernel.

To program the PL on the ZYNQ Z7020, we employ XILINX VIVADO 2015.1, along with the following XILINX IP cores: AXI BRAM Controller 4.0, AXI Central DMA 4.1, AXI Protocol Converter 2.1, Block Memory Generator 8.2, and ZYNQ Processing System 5.5.

To compile kernels for EPIPHANY, we use e-gcc 4.8.2 with flags `-O2 -mcmove -mno-soft-cmpsf -mfp-mode=truncate -mshort-calls -funroll-loops`. To link kernels for EPIPHANY, we wrote a custom linker script that only places code and constants used in time-critical sections of the kernel in on-core memory and everything else in a shared memory region on the SDRAM, to save on-core memory.

B. Image Quality

We use two metrics to compare a reflectivity map computed by our implementation to one computed by a double-precision reference implementation in MATLAB: the mean squared error (MSE) and the peak signal-to-noise ratio (PSNR).

The MSE measures the mean of the squared absolute point-wise differences of two reflectivity maps:

$$\mathrm{MSE}(A,B) := 10\log_{10}\left(\frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}|A_{i,j}-B_{i,j}|^2\right), \qquad (5)$$

where $A,B \in \mathbb{C}^{M\times N}$. A *smaller* MSE means that two images are closer to each another. This measure does not respect any human perception effects [28].

The PSNR relates the MSE to the maximum absolute value of a single point in the reflectivity map:

$$\mathrm{PSNR}(A,B) := 10\log_{10}\left(\frac{\max_{i,j}(A_{i,j})^2}{\mathrm{MSE}(A,B)}\right), \qquad (6)$$

Φ_{rm} [1/s]	Φ_{fp} [kFP/s]	Φ_{bf} [MBOP/s]	P_{proc} [W]	$P_{\mathrm{proc}}/\Phi_{\mathrm{rm}}$ [mJ]
5.27	194	24.8	2.1	399

Table II: Figures of merit for processing rate and energy consumption.

where $A,B \in \mathbb{C}^{M\times N}$ and A is the reference result. A *higher* PSNR means that two images are closer to each another. This measure accommodates the fact that small errors compared to the (maximum) signal amplitude are less perceptible than huge errors [29].

In Fig. 11, we visually compare the reflectivity map with the highest MSE. Receiver traces were captured with an Ultrasonix SonicTOUCH 128-channel device with a 5 MHz linear array. We measure a series of 128 shots and obtain an MSE of 1.27 dB and a PSNR of 41.55 dB. We conclude that the quality of the resulting images is sufficient, with full details visible on all images produced by our implementation: First, both MSE and PSNR are in the usual range for hardware implementations [4]. Second, when displayed on a screen with a contrast depth in the range of 8 .. 10 bit, the combined influence of our approximations will mostly be below the threshold of human perception.

C. Image Processing Rate and Energy Consumption

Having verified the output image quality of our implementation, we measure its processing performance and energy consumption. Power measurements were performed using a calibrated wall plug power measurement tool. Timing measurements were taken using counters on the host processor that were started just before the algorithm began its execution (i.e., after memory allocation and initialization, core resets, and kernel loading, but before any data transactions) and stopped as soon as the last result transaction was complete. One measurement consisted of consecutive beamforming of 128 shots; we averaged 10 such measurements.

We measured a power consumption of $P_{\mathrm{load}} = 7.1\mathrm{W}$ under full load and a processing time of $t_{\mathrm{proc}} = 24.35\mathrm{s}$. When idle, we measured $P_{\mathrm{idle}} = 5.0\mathrm{W}$. The figures of merit shown in Table II are derived as follows.

To specify the image processing rate, we calculated the *reflectivity map rate* $\Phi_{\mathrm{rm}} = t_{\mathrm{proc}}/128$, where one reflectivity map is computed from the 128×288 partial maps of 128 receivers. Other common figures of merit of the processing rate of a beamformer are the *focal point rate* $\Phi_{\mathrm{fp}} = 128 \cdot 288 \cdot \Phi_{\mathrm{rm}}$ and the *beamforming operation rate* $\Phi_{\mathrm{bf}} = 128 \cdot \Phi_{\mathrm{fp}}$, where this factor 128 corresponds to the number of receivers.

To specify the energy consumption of our beamformer, we have measured the idle power of the platform, P_{idle}, and the power under full load, P_{load}. We define the *processing*

Component	**Slice LUTs**		**Slice Registers**		**BRAM Tiles**		**DSP Slices**	
	abs.	rel.	abs.	rel.	abs.	rel.	abs.	rel.
Available	53200		106400		140		220	
ApoAccu incl. CDMA	2394	4.50 %	2586	2.43 %	0	0.00 %	16	7.27 %
Accumulation buffers	735	1.38 %	612	0.58 %	24	17.14 %	0	0.00 %
ELINK	1774	3.33 %	3188	3.00 %	0	0.00 %	0	0.00 %
Input buffers	2094	3.94 %	1652	1.55 %	96	68.57 %	0	0.00 %

Table III: Resource usage of the PL on the ZYNQ *Z7020. BRAM tiles are RAMB36, DSP slices are DSP48E1. Percentage values are relative to the amount of resources available for a given type.*

power $P_{\text{proc}} := P_{\text{load}} - P_{\text{idle}}$ to distinguish the dynamic power consumption of our implementation from the static power consumption of the platform, which contains many components unused in beamforming, e.g., an Ethernet interface, a USB controller, and an SD card interface. The *energy per reflectivity map* then follows from $P_{\text{proc}}/\Phi_{\text{rm}}$. It is indicative of the energy efficiency of the system.

Our evaluated implementation computes 5.27 reflectivity maps per second at 2.1 W of processing power, which corresponds to 399 mJ per computed reflectivity map. We conclude that our implementation is suitable to be applied in a low-power mobile beamforming system. We will address scaling to real-time imaging in Section VII-E.

D. Component Utilization

Table III shows the resource usage of the PL of the ZYNQ Z7020 by our apodization and accumulation implementation (*ApoAccu*, including a central DMA (CDMA) engine), by the accumulation buffers, by the ELINK interface implementation, and by the input buffers. Our implementation uses the DSP slices of the PL economically, i.e., just enough of them to reach the computing throughput required in order not to curtail the previous stage on the EPIPHANY. We clock *ApoAccu* at 100 MHz. The BRAM slices, on the other hand, are heavily utilized: one part to buffer intermediate results in accumulation, the other part to buffer data coming from the EPIPHANY. The input buffers necessarily are large to cover the latency of off-chip data transfers through the ELINK implementation on the PL. The accumulation buffers, on the other hand, are dimensioned to reduce the number of separate output data transfers; if more memory were required by other components, these buffers could be reduced in size at the cost of more output data transfers. Since, according to the heterogeneous parallel pipeline architecture, we map only tasks with simple control flow to the PL, the usage of LUT and register slices is low.

Figure 12 shows our memory layout for the beamforming kernel running on the coprocessor cores: Instructions use the majority of the first bank, the buffer for interpolated and modulated samples uses the majority of the second and the third bank, and the I/O buffers use the majority of the last bank; the rest of the last bank is allocated to the stack. Overall, 99.4 % of the memory of each core is used, with only 204 B of memory distributed over two banks remaining. Furthermore, our memory layout takes the frequently concurrent accesses by DMA engines, instruction sequencer, and load/store unit into account. Indeed, as we show below, there are little load/store stalls due to memory accessing conflicts.

We profiled our implementation on each ECORE with integrated timers. For this purpose, we divided the beamforming kernel into five sections: waiting for receiver data (`waitRxData`), interpolation and modulation (`InterpMod`), waiting for synchronization with consumer (`waitWrite`) and producer (`waitRead`), and sample selection with group-wise accumulation (`SelAccu`). We then performed multiple runs of 128 shots each, measuring clock cycles and different stall events. Table IV shows the

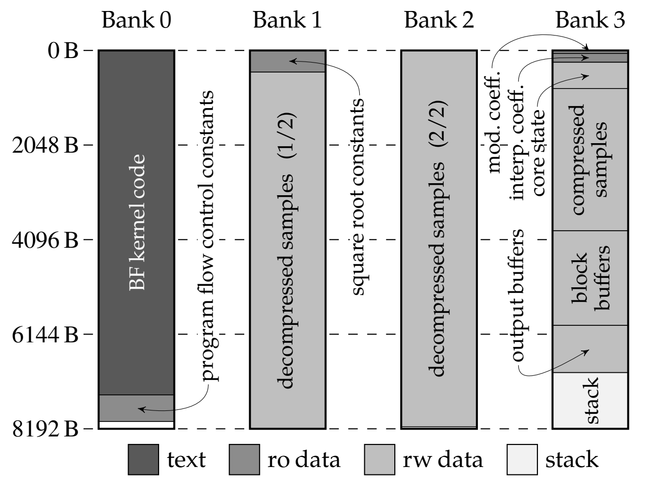


Figure 12: Memory layout of the beamforming kernel on a coprocessor core.

Section	**Clock Cycles**	**Share of Total Clocks**	**Execution Time**
`waitRxData`	1.39×10^6	0.02 %	2.32 ms
`InterpMod`	124.04×10^6	0.86 %	206.73 ms
`waitWrite`	1.01×10^6	0.01 %	1.68 ms
`waitRead`	0.87×10^6	0.01 %	1.44 ms
`SelAccu`	14767.91×10^6	99.10 %	24613.18 ms
Total	14895.22×10^6	100.00 %	24825.35 ms

Table IV: Execution time of the sections of the beamforming kernel on EPIPHANY *for* 128 *consecutive shots.*

Section		Total Clock Cycles	Register Access Stalls	E1 Stalls	Instruction Fetch Stalls	Load/Store Stalls
`InterpMod`	abs. $[10^6]$	124	8.86	5.07	5.06	0.01
	rel.		7.15 %	4.09 %	4.08 %	0.01 %
`SelAccu`	abs. $[10^6]$	14768	2412	952	64.9	5.09
	rel.		16.3 %	6.44 %	0.44 %	0.03 %

Table V: Stalls in the RISC pipeline of a coprocessor core in the compute sections of the beamforming kernel on EPIPHANY *for* 128 *consecutive shots. Percentage values are relative to the number of clock cycles spent in the section.*

amount of clock cycles spent in each section. Execution time is clearly dominated by `SelAccu`, taking nearly 120 times longer than `InterpMod`. We attribute the fact that waits take only 0.04 % of the total execution time to two design principles: First, our heterogeneous parallel pipeline allows to efficiently balance the load among the parallel cores of EPIPHANY. Second, our synchronization and buffering scheme introduces very little run-time overhead. Table V shows the reasons for stalls in the RISC pipeline of the core over the two computational sections. Most stalls are due to data dependencies, either in the register access or in the execution (E1) stage of the EPIPHANY core pipeline. While there are stalls due to conflicting accesses on local memory banks, their share is small (for instruction fetches) or even negligible (for data loads). We conclude that the RISC pipeline in the coprocessor cores is used efficiently and that our on-core memory layout leads to a low number of conflicting accesses.

In a separate set of measurements, we measured the coprocessor DMA engine transfer times. We have found that output data transfers from the coprocessor consume approximately 45 % of the time they could maximally take without stalling calculations. Thus, there is leeway, e.g., for chaining two EPIPHANY chips together or for using an EPIPHANY with more cores to obtain higher compound performance, as we discuss next.

E. Extrapolation to Real-Time Imaging

Our results demonstrate that the implementation is efficient in terms of resource usage and power consumption. However, we also showed that a 16-core EPIPHANY together with PL on another chip is not sufficient to perform real-time imaging. Using a 16-core EPIPHANY chip, our implementation achieves 5.27 reflectivity maps per second, whereas 80 such maps are required per second to meet our target frame rate, as discussed in Section III. However, the EPIPHANY architecture is available as scalable intellectual property block (IP block), and one possible configuration is the EPIPHANY E256G4 with 256 cores. A future platform could thus co-integrate the E256G4 IP block and programmable logic in a system-on-chip (SoC). Such a design is in line with those multiprocessor SoCs (MPSoCs) that co-integrate multi-core processors with programmable logic [30] and GPU IP blocks [17]. With the linear performance scaling the EPIPHANY architecture was designed for, we would be able to compute 84.23 reflectivity maps per second, which is sufficient to achieve the target frame rate. Furthermore, we estimate that, on a co-integrated platform, 16 of our PL *ApoAccu* blocks would fit on PL of the size found on a ZYNQ Z7030, for three reasons: First, the *ApoAccu* block scales linearly in the DSP slices to 16 instances, occupying about 64 % of the DSP slices of the Z7030, and the accumulation BRAMs can be shared by all accumulation units. Second, the eLINK interface, which would be replicated four times to connect 16×16 instead of 4×4 cores on one side of the grid, would consume about 10 % of the slices of the Z7030. Third, the large input BRAMs, which we use to cover the latency of transferring data from off-chip cores, could be massively reduced in a co-integrated solution to fit into the remaining BRAM slices.

VIII. CONCLUSION

In this work, we showed how to use heterogeneous multi-core platforms (particularly, platforms with multi-core coprocessors and programmable logic accelerators) to efficiently implement an ultrasound beamforming algorithm. The clear focus was on achieving sufficient performance with minimal power consumption. The main challenges lay in the severely limited on-chip memory capacities of the coprocessor in combination with the data bandwidth limitations between the coprocessor and the programmable logic but also to the external memory. Also, the question of how to map individual tasks to different computation resources was discussed.

We showed three methods to tackle these problems: First, we introduced a pipeline with internally parallel stages that are distributed across heterogeneous processing components to guarantee successful exploitation of all different computation resources with their individual capabilities at the cost of only small overhead. Second, we demonstrated various techniques for reducing the memory requirements on the coprocessor chip based both on live, on-demand computation of coefficients and on algorithmic considerations. Third, we reduced the effective computation workload by applying appropriate approximation techniques.

Experimental results on the ADAPTEVA PARALLELLA platform show that with only one EPIPHANY 16-core coprocessor chip and a small amount of programmable logic, 5.27 reflectivity maps of 128×288 pixels can be reconstructed per second. The dynamic power consumption is as low as 2 Watt, and synchronization overheads are below 0.05 %. Approximation errors are virtually not visible on the generated images. Scaling the architecture to a 256-core EPIPHANY IP block co-integrated on a single SoC with the programmable logic resources found on a ZYNQ Z7030 would allow real-time mobile ultrasound imaging.

ACKNOWLEDGMENT

This work was supported in part by the UltrasoundToGo RTD project (no. 20NA21 145911), evaluated by the Swiss NSF and funded by Nano-Tera.ch with Swiss Confederation financing.

BIBLIOGRAPHY

[1] J. Jensen *et al.*, "Performance of SARUS: A synthetic aperture real-time ultrasound system," in *IEEE Symposium on Ultrasonics*, Oct. 2010, pp. 305–309.

[2] J. Jensen *et al.*, "SARUS: A synthetic aperture real-time ultrasound system," *IEEE Trans. Ultrason., Ferroelectr., Freq. Control.*, vol. 60, no. 9, pp. 1838–1852, Sep. 2013.

[3] J. Park *et al.*, "Efficient implementation of a real-time dynamic synthetic aperture beamformer," in *2012 IEEE International Ultrasonics Symposium*, Oct 2012, pp. 2250–2253.

[4] P. A. Hager, A. Bartolini, and L. Benini, "Ekho: A 30.3W, 10k-channel fully digital integrated 3-D beamformer for medical ultrasound imaging achieving 298M focal points per second," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 5, pp. 1936–1949, May 2016.

[5] J. Ma *et al.*, "Software-based ultrasound phase rotation beamforming on multi-core DSP," in *2011 IEEE International Ultrasonics Symposium*, Oct 2011, pp. 503–506.

[6] SuperSonic Imagine, Aixplorer. www.supersonicimagine.com.

[7] B. Y. S. Yiu, I. K. H. Tsang, and A. C. H. Yu, "GPU-based beamformer: Fast realization of plane wave compounding and synthetic aperture imaging," *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, vol. 58, no. 8, pp. 1698–1705, August 2011.

[8] A. Tretter *et al.*, "Deterministic memory sharing in Kahn process networks: Ultrasound imaging as a case study," in *Embedded Systems for Real-time Multimedia (ESTIMedia), 2014 IEEE 12th Symposium on*, Oct 2014, pp. 80–89.

[9] J. D. Owens *et al.*, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

[10] J. Reinders, "An overview of programming for Intel Xeon processors and Intel Xeon Phi coprocessors," Tech. Rep., 2012.

[11] B. D. de Dinechin *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *High Performance Extreme Computing Conference (HPEC), 2013 IEEE*, Sept 2013, pp. 1–6.

[12] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," *arXiv preprint arXiv:1412.5538*, 2014.

[13] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, Jan 1998.

[14] J. E. Stone, D. Gohara, and G. Shi, "OpenCL: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[15] NVIDIA, "White paper: NVIDIA's next generation CUDA compute architecture," Tech. Rep.

[16] A. Varghese *et al.*, "Programming the Adapteva Epiphany 64-core network-on-chip coprocessor," *CoRR*, vol. abs/1410.8772, 2014.

[17] Xilinx, "White paper: Unleash the unparalleled power and flexibility of Zynq UltraScale+ MPSoCs," Tech. Rep., Jun. 2016.

[18] I. Berkeley Design Technology, "An independent evaluation of high-level synthesis tools for Xilinx FPGAs," Tech. Rep., 2010.

[19] P. Vogel, A. Marongiu, and L. Benini, "Lightweight virtual memory support for many-core accelerators in heterogeneous embedded SoCs," in *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2015 International Conference on*, Oct 2015, pp. 45–54.

[20] J. A. Jensen *et al.*, "Synthetic aperture ultrasound imaging," *Ultrasonics*, vol. 44, Supplement, pp. e5 – e15, 2006, proceedings of Ultrasonics International (UI'05) and World Congress on Ultrasonics (WCU).

[21] W. Thies, M. Karczmarek, and S. Amarasinghe, "StreamIt: A language for streaming applications," in *International Conference on Compiler Construction*. Springer, 2002, pp. 179–196.

[22] L. Schor *et al.*, "Scenario-based design flow for mapping streaming applications onto on-chip many-core systems," in *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*. ACM, 2012, pp. 71–80.

[23] J. Ceng *et al.*, "MAPS: An integrated framework for MPSoC application parallelization," in *Proc. Design Automation Conference (DAC)*, 2008, pp. 754–759.

[24] W. J. Cody, *Software Manual for the Elementary Functions (Prentice-Hall Series in Computational Mathematics)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1980.

[25] P. Vogel, A. Bartolini, and L. Benini, "Efficient parallel beamforming for 3D ultrasound imaging," in *Proceedings of the 24th Edition of the Great Lakes Symposium on VLSI*. ACM, 2014, pp. 175–180.

[26] I.-C. Park and T.-H. Kim, "Multiplier-less and table-less linear approximation for square and square-root," in *Computer Design, 2009. ICCD 2009. IEEE International Conference on*. IEEE, 2009, pp. 378–383.

[27] *Epiphany SDK Reference*, Adapteva, 2013.

[28] Z. Wang *et al.*, "Image quality assessment: from error visibility to structural similarity," *IEEE Transactions on Image Processing*, vol. 13, no. 4, pp. 600–612, April 2004.

[29] D. Salomon, *Data Compression: The Complete Reference*. Springer New York, 2000.

[30] Altera, "White paper: Architecture matters: Choosing the right SoC FPGA for your application," Tech. Rep., Nov. 2013.