Radio Astronomy Beam Forming on Many-Core Architectures

Alessio Sclocco, Ana Lucia Varbanescu
Faculty of Sciences
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
a.sclocco@vu.nl, a.l.varbanescu@vu.nl

Jan David Mol, Rob V. van Nieuwpoort
ASTRON
Netherlands Institute for Radio Astronomy
Dwingeloo, The Netherlands
mol@astron.nl, nieuwpoort@astron.nl

***Abstract*—Traditional radio telescopes use large steel dishes to observe radio sources. The largest radio telescope in the world, LOFAR, uses tens of thousands of fixed, omni-directional antennas instead, a novel design that promises ground-breaking research in astronomy. Where traditional telescopes use custom-built hardware, LOFAR uses *software* to do signal processing in real time. This leads to an instrument that is inherently more flexible. However, the enormous data rates and processing requirements (tens to hundreds of teraflops) make this extremely challenging. The next-generation telescope, the SKA, will require exaflops. Unlike traditional instruments, LOFAR and SKA can observe in hundreds of directions simultaneously, using *beam forming*. This is useful, for example, to search the sky for pulsars (i.e. rapidly rotating highly magnetized neutron stars). Beam forming is an important technique in signal processing: it is also used in WIFI and 4G cellular networks, radar systems, and health-care microwave imaging instruments.**

We propose the use of many-core architectures, such as 48-core CPU systems and Graphics Processing Units (GPUs), to accelerate beam forming. We use two different frameworks for GPUs, CUDA and OpenCL, and present results for hardware from different vendors (i.e. AMD and NVIDIA). Additionally, we implement the LOFAR beam former on multi-core CPUs, using OpenMP with SSE vector instructions. We use auto-tuning to support different architectures and implementation frameworks, achieving both platform and performance portability. Finally, we compare our results with the production implementation, written in assembly and running on an IBM Blue Gene/P supercomputer. We compare both computational and power efficiency, since power usage is one of the fundamental challenges modern radio telescopes face. Compared to the production implementation, our auto-tuned beam former is 45–50 times faster on GPUs, and 2–8 times more power efficient. Our experimental results lead to the conclusion that GPUs are an attractive solution to accelerate beam forming.

***Keywords*-radio astronomy, many-core architectures, beam forming, auto tuning**

I. Introduction

Radio telescopes are used to capture the frequencies of the electromagnetic spectrum that are outside the realm of visible light. To answer astronomy's big open questions, for instance those related to the origin of the universe, there is a need for more powerful and precise instruments. Due to engineering limitations and economical constraints, simply building larger telescopes to increase resolution is not viable anymore. An alternative is radio interferometry, a technique that combines signals of multiple receivers, thus creating a single large virtual telescope. To cope with the complexity of this scenario, telescopes rapidly evolve into *software telescopes*, while they used hardware-based processing in the past. The software solution increases flexibility and lowers construction costs. The computational demands are challenging: LOFAR requires tens to hundreds of teraflops. The SKA, a next-generation instrument that is currently being designed, will require exaflops. Therefore, high performance and power efficiency are of key importance.

LOFAR is an example of such a software telescope: it has more than 80,000 omni-directional antennas which are geographically distributed in five different countries. LOFAR's signal processing pipeline is implemented in software, and runs on an IBM Blue Gene/P (BG/P) supercomputer. LOFAR is the largest and most complex radio telescope in the world. It is driven by the astronomical community, which needs a new instrument to study an extensive amount of new science cases. Five key science projects have been defined. First, we expect to see the *Epoch of Reionization* (EoR), the time when the first star galaxies and quasars were formed. Second, LOFAR offers a unique possibility in particle astrophysics for studying the origin of high-energy *cosmic rays*. Neither the source, nor the physical process that accelerates such particles is known. Third, LOFAR's ability to continuously monitor a large fraction of the sky makes it uniquely suited to find new *pulsars* and to study *transient sources*. Fourth, *Deep Extra-galactic Surveys* will be carried out to find the most distant radio galaxies and study star-forming galaxies. Fifth, LOFAR will be capable of observing the so far unexplored radio waves emitted by *cosmic magnetic fields*. For a more extensive description of the astronomical aspects of the LOFAR system, see [1].

One of LOFAR's key features is the capability to point at multiple directions in the sky at the same time, without the need to move any mechanical part; this is made possible by its software *beam former*. In fact, beam forming is *the only way* to point the LOFAR telescope. With the high number of antennas, and the ever increasing number of observations requested by the astronomers for their experiments, it becomes apparent that the beam former is a fundamental

part of the telescope's pipeline, with particularly demanding requirements of high performance and scalability.

LOFAR's production beam former is currently executed on a Blue Gene/P, together with many other components of the LOFAR pipeline. The beam former is an inherently parallel application. This paper investigates whether or not it is possible and effective to parallelize the beam former using modern many-core architectures, such as *Graphics Processing Units* (GPUs), and large multi-core CPU nodes. Our aim is to achieve high performance and power efficiency.

We are interested in GPUs as a possible way to accelerate radio astronomy because in the last ten years they have evolved from simple graphics processors to general purpose computational units, offering a mix of low costs, high computational power, high memory bandwidth, and low power consumption. To verify if they are a viable solution, we have redesigned, implemented and optimized the LOFAR beam former for the NVIDIA GTX580 and the AMD HD6970 video cards, using the *Compute Unified Device Architecture* (CUDA) [2] and the *Open Computing Language* (OpenCL) [3] as implementation frameworks. OpenCL is a portable platform, and multi-core CPUs can also execute the OpenCL beam former. An important question is if OpenCL also provides performance portability. Therefore, on CPUs, we compare with an additional alternative implementation using OpenMP and manual SSE vectorization.

All versions exploit our auto-tuning and run-time code generation techniques to adapt the implementation to the hardware platform and problem parameters, as dictated by the observation specification. To gain insights in performance and power efficiency improvements, we compare our novel beam former with LOFAR's production version on the BG/P supercomputer, which is hand-written in assembly, and extremely efficient. Our results indicate that our auto-tuning many-core code is up to 50 times faster, and 3 times more power efficient on GPUs. Moreover, using 8 GPUs in a single node, our beam former even is 8 times more power efficient than the BG/P. This excellent result will allow us to build larger telescopes, and to point in more directions simultaneously, leading to more effective instruments.

The rest of this paper is organized as follows. First, we summarize the current state in the field of beam forming for radio astronomy in Section II. Then, we present background on telescopes, with a focus on LOFAR, and beam forming, in Sections III and IV respectively. In Section V we analyze the LOFAR beam former, moving from the sequential to the GPU algorithm, while introducing the most important optimization strategies used. The auto-tuning of the beam former for the different architectures is introduced in Section VI. In Section VII we present a detailed analysis of the algorithm's performance and power efficiency on different many-core platforms, while comparing to the production version. Finally, Section VIII presents our conclusions.

II. Related Work

So far, there are few software beam formers in use in astronomy, as beam forming is an operation that is still implemented in hardware in the vast majority of cases. Among the software implementations, the one that is more relevant for our work is the LOFAR production beam former, described in [4]. An overview of the real-time software pipeline of LOFAR is presented in [5]. Another software beam former is the one of the *Giant Metrewave Radio Telescope* (GMRT) in India [6]. This beam former, running on a cluster of commodity hardware, produces at most 32 dual polarized beams, while the LOFAR beam former is capable of producing hundreds of different beams: this clearly shows how difficult the operational challenges of LOFAR are. It is interesting to observe that also for the GMRT the use of GPUs is considered as a possible future solution to improve performance and to cut costs [6].

A different approach to beam forming is the one of OSKAR [7]: a research tool, developed by the Oxford astrophysics and e-Research groups, used to investigate the challenges of beam forming for the SKA radio telescope. It currently supports two different modes of execution: the simulation of the beam forming phase and the computation of different beam patterns. Due to the fact that OSKAR is a simulation framework, it is not possible to directly compare its approach and performance with ours.

As far as we know, there are no GPU radio astronomy beam formers at the moment. There are, however, some attempts at implementing general domain beam formers using GPUs. Nilsen et al. [8] present two different digital beam formers, implemented with CUDA, using an NVIDIA GeForce 8800. The authors conclude that they can see a future for the use of GPUs as the platform to run digital, high performance, beam formers. The hardware that we use in this paper is significantly different, leading to different trade-offs. Also, our implementation is portable across different platforms thanks to the use of OpenCL and auto-tuning.

Beam forming is a general signal processing technique, and software beam formers are used in many different areas. For example, [9] presents two beam forming techniques aimed at increasing the number of possible users of an I-WiMAX maritime communication system. Beam forming is an efficient solution to reduce the spectrum necessary for wireless communication because the waves can be steered to the direction of the receiver, thus reducing interferences. Therefore, beam forming techniques are used in modern WIFI and 4G cellular telephone networks (TLE).

Another example of the importance of beam forming can be found in [10]. Here, the authors improve the flexibility of a radar system used to monitor the ocean, using a software beam former. The use of this kind of beam former permits to deploy this type of radar systems in locations that were not suitable before. Also, the field of medicine benefits from

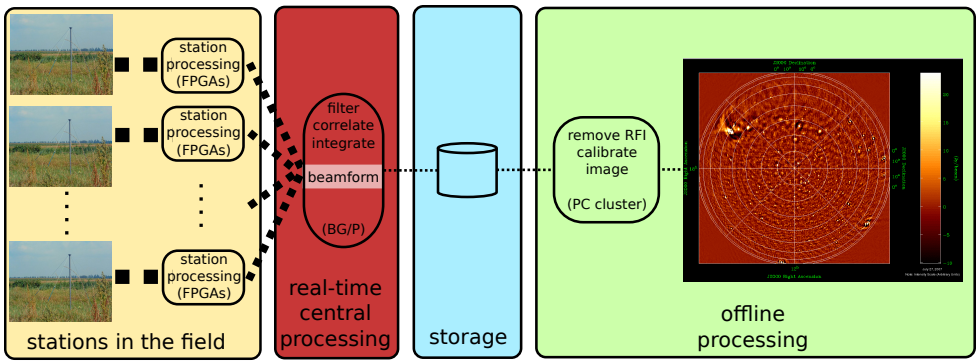

Figure 1. A simplified overview of LOFAR processing.

software beam forming. In [11], different beam forming algorithms are compared to find which one is the best for breast cancer detection using microwave imaging.

III. Software Telescopes

The structure of a classic radio telescope is relatively simple: first, a large metallic dish reflects the radio waves to an electronic receiver in its focal point. Next, the received signals are processed, usually by special-purpose hardware, and transformed into a representation that the astronomers can use. In the past, the need to improve precision and sensitivity of telescopes has always been addressed by building bigger dishes. However, this is a solution that does not scale: there are physical and engineering constraints on the dimensions that a dish may reach. In addition, moving a huge metal dish to point at some specific direction in the sky is slow and difficult. This is a severe limitation for several science cases, such as transient detection. Moreover, building a massive telescope of this kind implies enormous costs for raw materials and realization.

An attractive alternative is provided by radio interferometry, a technique that combines the signals received by different antennas into a single and meaningful signal. It is thus possible with this technique to use small dishes or antennas (simpler and cheaper than large metal dishes) and combine their measurements to build an instrument with a resolution that is equivalent to that of a telescope with a diameter equal to the largest distance between antennas. Such telescopes essentially are distributed sensor networks.

However, a distributed telescope poses its own challenges: it is necessary to connect all receiving antennas to a central processing unit, and perform operations on the recorded samples to merge (correlate) them. Additionally, since the data streams are too large to store on disk, all these operations must be done in real-time. Implementing a system like this completely in hardware is costly and error prone, and may even span a period of over a decade to go from the design to the realization. Moreover, it requires expertises that are not easily found on the market. Most importantly though, hardware implementations lack flexibility: a telescope has a long lifetime (decades) during which there is a high probability that new requirements will arise, and that the operational setup will change. What appears to be the best solution for providing flexibility for the new generation of telescopes is to implement their operational pipelines in software.

In this paper, we use the LOFAR telescope as a driving example. LOFAR is a large radio interferometric array, and a perfect example of what we call a software telescope: it currently is the largest radio telescope in the world, with more than 80,000 antennas, and its software pipeline is executed in real-time by a two and a half rack IBM Blue Gene/P supercomputer. The antennas, all of them omni-directional, are of two different types: low-band antennas, for the frequency interval of 10-80 MHz, and high-band antennas, for the 110-240 MHz range. These antennas are not directly connected to the central computing facility, instead they are co-located in groups of different dimensions, and organized in *stations*. There are 20 core stations, all situated together in the northern part of the Netherlands, and 24 external stations at increasing distance from the core. Each station is equipped with a cabinet where some preliminary processing is performed. Each core station may act as two different stations, bringing the number of LOFAR stations to 64. Stations are important because they increase scalability: the software at the central computing facility can use the stations' output as its input, instead of dealing with each single antenna.

Figure 1 shows an overview of the LOFAR processing pipeline. The left part, before the storage, is executed in real time. This part includes the beam former. After beam forming, data is stored to disk, and processed further off-line. For more detailed information on the pipeline, we refer to [5]. In the next section, we will describe the beam forming process in more detail.

IV. Beam Forming

Beam forming is a standard signal processing technique that is used to control the spatial selectivity of omni-

directional antennas. It is important in radio interferometry, because the signals from the receiving antennas need to be compensated for the different antenna positions. This is shown in Figure 2: all antennas receive the signals from a radio source at different moments in time. Signals are compensated before being integrated (added together), by shifting their phase and amplitude by a value that depends on the position of the source and of the antennas. Without this compensation, the formed beams would have no directionality. In case of narrow-band systems, such as LOFAR, the amplitude shift is unnecessary, and *a phase shift is sufficient to provide the correct compensation*.

LOFAR has two main software beam formers: the superstation and the tied-array beam former. The superstation beam former is part of the telescope's imaging pipeline. It reduces the number of stations seen by the correlator (i.e. the component that computes sky images [5]), thus lowering its complexity. This beam former simply adds samples together, without shifting them. We focus on the tied-array beam former, since it provides directionality in LOFAR.

The input samples of the beam former are grouped in *channels*. A channel represents an observation's frequency interval (in our case of 763 Hz). For this interval, the input contains all samples, for all used stations, measured in two polarizations (X and Y). The output consists of an arbitrary number of beams, that may vary between few and many hundreds, with each beam representing a different pointing direction. A beam contains all the frequency channels, and their samples in both polarizations. The complexity of the beam former is $O(s \cdot b)$, where s is the number of input stations and b is the number of generated output beams.

Beam forming is especially important for the detection of transient objects. Thanks to forming multiple beams at the same time, and thus pointing at many directions simultaneously, it is possible to use the telescope for monitoring a large fraction of the sky and wait for unexpected events. Moreover, if another instrument detects an object outside the area of the sky that LOFAR is currently monitoring, the software beam former can be reprogrammed to redirect its focus in real-time, without the need to move any mechanical part. We refer to [4] for more information on how beam forming is used in LOFAR.

V. Application Analysis

In this section, we first introduce the sequential beam forming algorithm. Subsequently, we present our multi-core and GPU parallel beam formers, explaining the parallelization and choice of optimization strategies.

A. The Sequential Algorithm

The LOFAR beam forming algorithm consists of three successive stages: (1) delays computation, (2) flagging bad samples, and (3) beam forming. The goal of the delays computation stage is to compute the time delays that are needed to correct for the different antenna positions and for pointing the instrument. These are different for each possible combination of stations and beams. The input of this stage are the delays at the beginning and the end of a samples period, for each pair of stations and beams. For each beam and station the algorithm computes an average of these two values, subtracts from it the precomputed delay of the first station, and stores the result in a matrix.

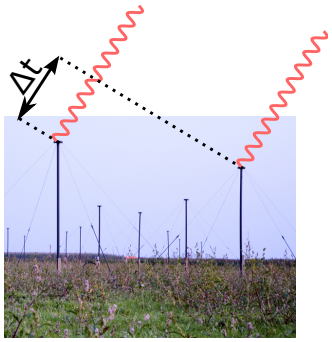

Figure 2. The rightmost antenna receives the signal earlier.

The second stage, flagging bad samples, is necessary because received signals may contain errors due to radio frequency interference (RFI). Since the beam former integrates data from different stations, errors from one station will propagate, and pollute all output beams. A station is invalid and excluded from the computation if the number of its flagged samples, i.e. the samples containing errors, is above a certain threshold. Even if a station is valid it may still contain some errors. To account for this, if a station's sample is flagged, the corresponding sample in all output beams is flagged as well.

Algorithm 1 Pseudocode for the third stage of the beam forming algorithm.

```
for b = 0 → nrBeams do
  for c = 0 → nrChannels do
    for t = 0 → nrTimes do
      for p = 0 → nrPolarizations do
        beam = 0
        for s = 0 → nrStations do
          sample = Input[c][s][t][p]
          shift = computeShift(c, s, b)
          sample = sample * shift
          beam = beam + sample
        end for
        Output[b][c][t][p] = beam/nrValidStations
      end for
    end for
  end for
end for
```

The third stage is the proper beam forming stage, and it accounts for most of the execution time. For each beam to form, the algorithm iterates over all three dimensions

(channels, samples and polarizations), loads each station's samples, phase shifts them, and outputs the average of these shifted samples. Algorithm 1 shows pseudocode for the main loop of this stage. The phase shift value is a function of the previously computed delay, and of the sample's frequency. All the signals are represented as single precision floating point complex numbers, so all the arithmetic operations in the pseudocode are in fact complex operations.

The delays computed in the first stage are used only to compute the phase shifts in the third stage. Therefore, we extract the phase shifts computation from the third stage and move it to the first. Thus, the *computeShift()* operation in Algorithm 1 simply becomes an access to a lookup table. As a side effect of this merge, the beam forming stage is simplified, and uses only single-precision floating point operations. In fact, the only operations now are complex additions and multiplications, which are efficiently implemented on all platforms. Even after the removal of the phase shifts computation, the third stage remains the most time consuming part of the algorithm.

Analysis of the sequential algorithm shows that the beam forming algorithm is inherently parallel: there are no data dependencies between different beams, and they can be computed independently of each other. We can affirm that the delays computation and the flagging stages don't have dependencies between them, thus they can be independently computed, possibly concurrently. The third stage, however, has to wait until the completion of the previous stages, as its computation depends on their outputs.

B. The IBM Blue Gene/P Production Version

The current production version of the LOFAR beam former runs on an IBM Blue Gene/P supercomputer. It is implemented in C++, with the core routines written in assembly for performance reasons. The code presented in Algorithm 1 is written in assembly and is manually tuned to minimize memory accesses and maximize hardware utilization. This highly optimized implementation of the beam former achieves 86% of the platform's peak performance. For a further description and performance analysis of the production beam former we refer to [4].

C. The Multi-core CPU Version

A beam former for multi-core CPUs needs to be flexible enough to leverage the ever increasing number of available cores per node. Moreover, modern CPUs provide different levels of parallelism as they also support SIMD instructions (SSE); exploiting all these levels of parallelism at the same time is critical to achieve high performance.

Given that the phase shifts are computed for each combination of channels, stations and beams, and that each phase shift is independent of all the others, we use OpenMP [12] to divide the work in the first stage between different threads. For each different channel a thread works on a non-overlapping subset of stations, and computes the phase shifts associated with these stations and all the beams.

In the flagging stage, we can use OpenMP to parallelize counting the valid stations and flagging the output. However, this part of the algorithm is less computation intensive and, consequently, benefits less from parallelization. Therefore, we do not further investigate the parallelization of this stage.

For stage three, we have two different levels of parallelism: (1) samples are equally divided between OpenMP threads and (2) the two polarizations of each sample are computed in parallel using the Streaming SIMD Extensions (SSE) [13]. The beam former kernel uses a different thread for each frequency channel and, for each channel, this thread spans a small number of children, each of them working on a part of the samples and being responsible for the computation of all beams, i.e. it merges all the shifted samples, in both polarizations, of all the different stations.

D. The GPU Version

Here we introduce the parallelization strategies behind our GPU algorithm; as common in high performance computing, these strategies are a direct consequence of the hardware organization of the platform. The described algorithm is the best performing one of a family of six different GPU beam forming algorithms that we designed and implemented in previous work [14]. To achieve good performance it is necessary to understand that GPUs are inherently hierarchical devices. In fact, they use two different hierarchical abstractions: the computational and the memory organization. From a computational point of view, GPUs are equipped with a variable number of streaming multiprocessors, each of which contains many computational cores. GPU computations are organized in thread-blocks and threads, with each thread-block being associated with a streaming multiprocessor, and each of the block's threads being executed by one of the streaming multiprocessor's cores. In addition, GPUs have several different memories. Most important are the off-chip global memory, accessible by all threads of all thread-blocks, and the on-chip shared memory, that is available only to the threads of a same block, and may be used as an application-controlled cache.

Data transfers from the host to the GPU over the PCI-e bus can be a bottleneck for data-intensive computations [15]. In radio astronomy signal processing, this has also been identified as a problem [16]. However, for this work, we do not take the host-GPU data transfers into account, since the beam former is a part of a larger pipeline [5]. Therefore, we assume the data already is on the GPU, and may also be used on the GPU again for further processing.

The delays and phase shift computations present a degree of parallelism that may benefit from a GPU implementation. However, they use double-precision floating point operations and trigonometric functions, both of which are expensive on

GPUs. We did investigate GPU parallelization of this stage, but the OpenMP/SSE CPU implementation performed better.

The flagging stage has less advantage of being parallelized on the GPU, since it has limited data parallelism and uses non-trivial data structures (i.e. C++ sparse sets). Moreover, of this stage's outputs, only the number of valid stations is directly used in the following phase, and this value can be easily passed as an argument to the GPU kernel.

The third phase of the algorithm benefits the most from parallelization on a GPU. In the design of our GPU beam former we exploit two levels of parallelism. In the first level, we assign each channel to a different thread-block. All samples are independent in the time direction, thus inside each thread-block we assign a different sample to every single thread. Each thread is then responsible for merging and shifting all stations and all beams, in both polarizations, but just for the channel and time associated with it. The advantage of this solution is that it eliminates the need for any inter thread, and inter thread-block communication: each thread can run independently from the others.

Moreover, this structure for the computation permits coalesced accesses to the GPU's global memory, as the threads inside a block read their inputs from, and write their outputs to, consecutive memory addresses. This is important because, on GPUs, coalescing is critical for performance [17]. Due to hardware or implementation framework limitations on some platforms the structure may be slightly modified at runtime, i.e. splitting each original thread-block into multiple blocks, but this does not modify the overall structure of the algorithm.

E. The beams-block Optimization Strategy

Sections V-C and V-D outlined how the beam forming algorithm can be parallelized. However, achieving good performance is still difficult. To achieve good performance on architectures where the gap between computational power and memory bandwidth is wide (such as on GPUs), it is extremely important to minimize the number of accesses to the slow global memory [16]. Minimizing memory accesses is even more important for algorithms with low arithmetic intensity (AI) [18], such as the beam former. Its arithmetic intensity, the number of operations performed per byte accessed in global memory, can simply be counted looking at the source code, and is shown in Equation 1.

$$AI = \frac{(4 \times stations) + 1}{(6 \times stations) + 4} \tag{1}$$

A way to minimize accesses to memory is to increase data reuse: when loading data from global memory, it is important to perform all, or most of, the operations associated with these data. There are two different points in our algorithm with potential for data reuse: (1) all the threads of a thread-block may share the phase shifts, because they are independent with respect to time, and (2) a single thread may reuse a loaded station's sample to compute many beams. To implement this optimization strategy we modified the kernel's main loop as shown in Algorithm 2.

Algorithm 2 Pseudocode for the kernel's beams-block optimization.

```
c = myChannel()
t = myTime()
for station = firstStation → nrStations do
    samplePolarization0 = Input[c][station][t][0]
    samplePolarization1 = Input[c][station][t][1]
    for beam = firstBeam → firstBeam + beamBlockDim do
        shift = Shifts[c][station][beam]
        beam0 = samplePolarization0 * shift
        beam1 = samplePolarization1 * shift
        Beams[beam][0] = Beams[beam][0] + beam0
        Beams[beam][1] = Beams[beam][1] + beam1
    end for
end for
```

The first crucial difference with Algorithm 1 is that the order of the loops over stations and beams is changed. This enables reuse of loaded station samples to form many beams. Another important difference is that the beams loop is not over the whole space of the beams, but only over a block.

This is necessary because it is not always possible to compute all the beams within a single kernel execution due to the limited amount of registers. Even though we use arrays in the pseudocode, the actual code uses registers instead of memory for performance reasons. Thus, the number of beams that is possible to form within a single execution is limited by the number of registers that are available per thread. Furthermore, in the source code the memory operations are vectorized, so the two polarizations are loaded and stored with a single operation.

We call the number of beams formed during an iteration of the innermost loop the *beams-block*. This parameter is of capital importance for the performance of the algorithm: a correct setup may effectively improve performance, while a wrong one may lead to hardware underutilization, a non optimal number of memory accesses if the block is too small, and register spilling if the block is too large. The performance improvement brought by the beams-block optimization strategy is also reflected in the arithmetic intensity, as shown in Equation 2. The kernel's arithmetic intensity increases with a larger beams-block size.

$$AI = \frac{(4 \times stations) + 1}{\frac{4 \times stations}{beams-block} + (2 \times stations) + 4} \tag{2}$$

VI. Auto-tuning the Beams-block

To provide a fair platform performance comparison, we need to select the best configuration of the beams-block parameter. To tune the algorithm, we try different configurations, and measure the number of single precision floating point operations per second (GFLOP/s) achieved in the third

Platform	Cores	GFLOP/s	GB/s	TDP (Watt)
IBM Blue Gene/P	4×1	13.6	13.6	24
Intel Xeon E5620	4×2	153.6	51.6	160
AMD Opteron 6172	12×4	806.4	170.4	320
AMD HD6970	64×24	2793	176	250
NVIDIA GTX580	32×16	1581.1	192.4	244

Table I
CHARACTERISTICS OF THE USED PLATFORMS.

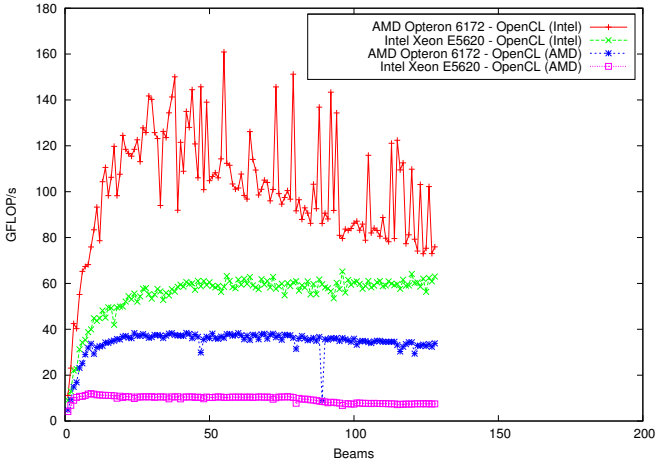

Figure 3. Comparison of Intel and AMD OpenCL compilers for CPU, 512 stations (higher is better).

stage. The analysis of these values provides general and case-specific guidelines for the setup of the beams-block.

The experiments are performed using the *Distributed ASCI Supercomputer 4* (DAS-4) [19], running CentOS Linux release 6. The C++ compiler used for the CPU code is the GNU g++, version 4.4.6, that implements OpenMP 3.1; we obtained slightly lower results accross the board using the Intel C++ compiler, version 12.1, thus we are not including it in the discussion. For running the OpenCL implementation on multi-core CPUs we rely on the runtime environment included with the Intel OpenCL SDK 1.1, as we found that for our application it is much faster than AMD's SDK (see Figure 3). For the NVIDIA GPU we use CUDA 4.0, that also provides an OpenCL runtime, while for the AMD GPU we use the *AMD Accelerated Parallel Processing* (APP) SDK version 2.5. Platform characteristics are summarized in Table I.

A. IBM Blue Gene/P

The LOFAR production code uses an optimization strategy based on combining multiple iterations of the beam former's main loop. After careful analysis and manual tuning we found that the optimal setup is to compute 128 time samples, 6 stations and 3 beams for each kernel iteration. The tuning of the production version of the beam former on the BG/P is beyond the scope of this work, thus we do not further discuss it and refer to [4] for more information.

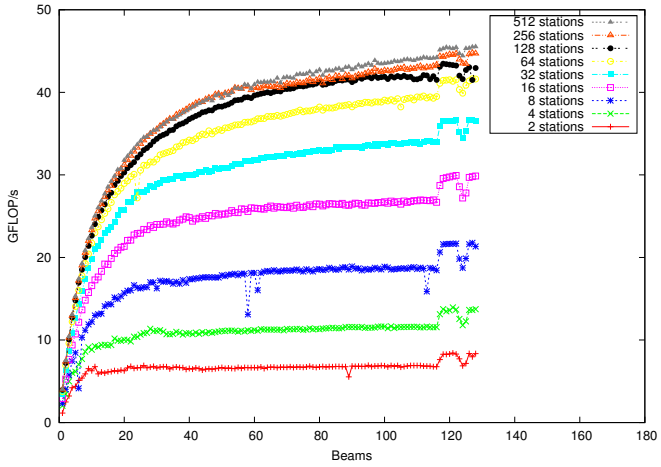

Figure 4. Tuning the beams-block for the Intel Xeon E5620 using OpenMP/SSE (higher is better).

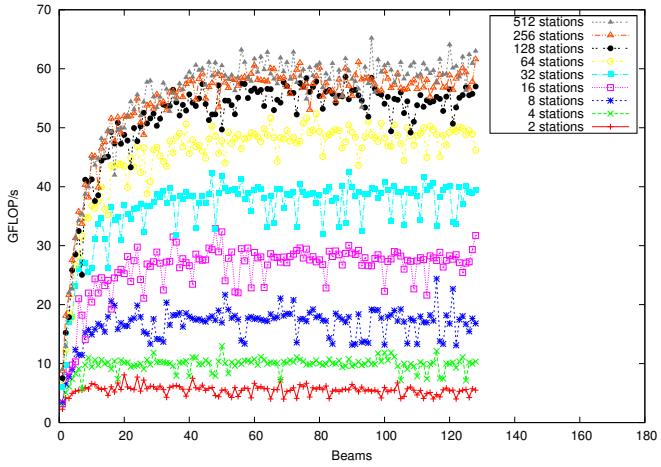

Figure 5. Tuning the beams-block for the Intel Xeon E5620 using OpenCL (higher is better).

B. Intel Xeon E5620

The first multi-core CPU we use is the Intel Xeon E5620. Figure 4 presents the results obtained using the OpenMP/SSE version (described in Section V-C). Results show that performance is increasing with the size of the beams-block. The returns are diminishing for higher values, however, because the beams-block size determines the amount of registers used, and for higher values, the compiler spills the registers to memory. Nevertheless, the best setup for this platform is to set the beams-block equal to the number of beams to form.

The Intel CPUs also support OpenCL, and we auto-tuned the OpenCL implementation (described in Section V-D) for this platform as well. Figure 5 shows the results for this experiment. Despite some outliers, the behavior of the OpenCL implementation is close to the OpenMP/SSE

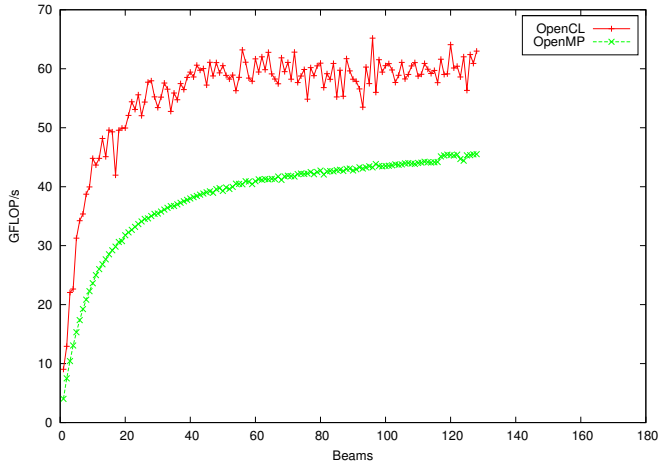

Figure 6. Comparison between OpenMP/SSE and OpenCL on the Intel Xeon E5620, 512 stations (higher is better).

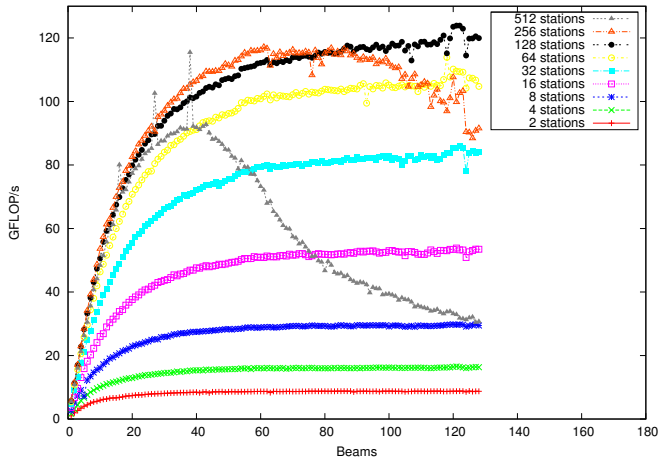

Figure 7. Tuning the beams-block for the AMD Opteron 6172 using OpenMP/SSE (higher is better).

version. With OpenCL, the best setup of the beams-block is also to use a value equal to the number of beams.

Since we have two different implementations on the same hardware, we compare the performance achieved on this platform by OpenMP/SSE and OpenCL in Figure 6. The comparison shows that the OpenCL implementation achieves higher performance than the OpenMP/SSE version: the OpenCL implementations achieves 65 GFLOP/s (42% of the platform's peak), while the OpenMP/SSE implementation achieves only 45 GFLOP/s (29% of the platform's peak). This result can be explained by further optimizations applied by the Intel's OpenCL compiler compared to GCC.

C. AMD Opteron 6172

The second multi-core CPU that we use is the AMD Opteron 6172. We run the same experiment that we previously described for the Intel CPU. Figures 7 and 8 present

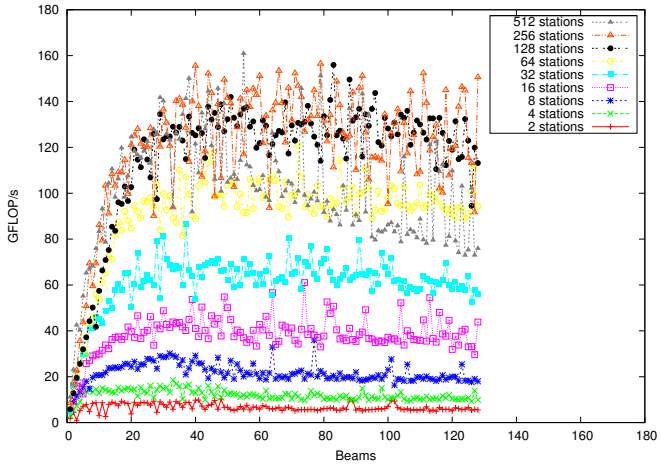

Figure 8. Tuning the beams-block for the AMD Opteron 6172 using OpenCL (higher is better).

the results obtained using OpenMP/SSE and OpenCL, respectively.

The OpenMP/SSE implementation behaves almost the same as on the Intel CPU, with the difference that on the Opteron CPU the curves are plateauing later, thanks to the higher number of available cores. We also see that for a high number of stations performance is not always increasing with the beams-block size, i.e. it rises to a peak and then there is a sudden drop. Thus, we can conclude that the best setup of the beams-block for this platform equal to the number of beams when there are not so many stations, and to select a smaller value if the number of stations is larger than 128. The specific value is input dependent and can be identified using the results of this auto-tuning step.

The behavior of the OpenCL implementation on the Opteron 6172 is far less stable than what observed on the Intel CPU, and even less stable than the OpenMP/SSE implementation on the same hardware. However, if we exclude the outliers, we see the same trend that we had with the OpenMP/SSE implementation, thus we can conclude the same for what concerns the size of the beams-block size.

Figure 9 shows a comparison of OpenMP/SSE and OpenCL for this platform. The OpenCL implementation again achieves more GFLOP/s than OpenMP/SSE: 161 against 123. However, our many-core beam former is less efficient on the AMD CPU, reaching only 20% and 15% of the theoretical peak, respectively. This is caused by the relatively low memory bandwidth *per core* of the AMD machine, which hurts our data-intensive code.

D. NVIDIA GTX580

The NVIDIA GTX580 card can run both CUDA and OpenCL. The number of stations varies between 2 and 512, while the number of beams varies between 1 and 16; we have this large difference in the maximum number of stations

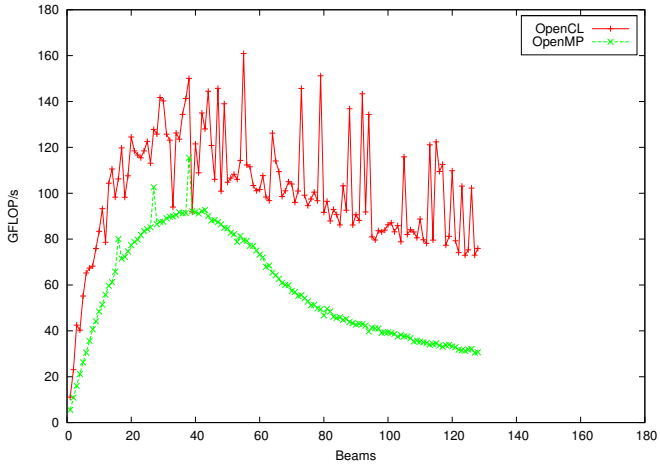

Figure 9. Comparison between OpenMP/SSE and OpenCL on the AMD Opteron 6172, 512 stations (higher is better).

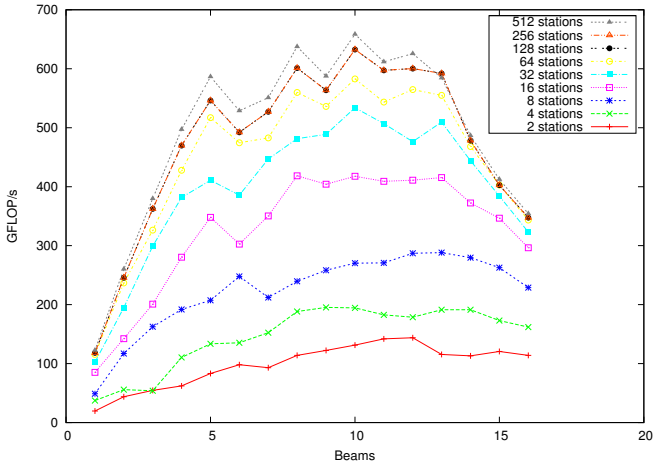

Figure 11. Tuning the beams-block for the NVIDIA GTX580 using OpenCL (higher is better).

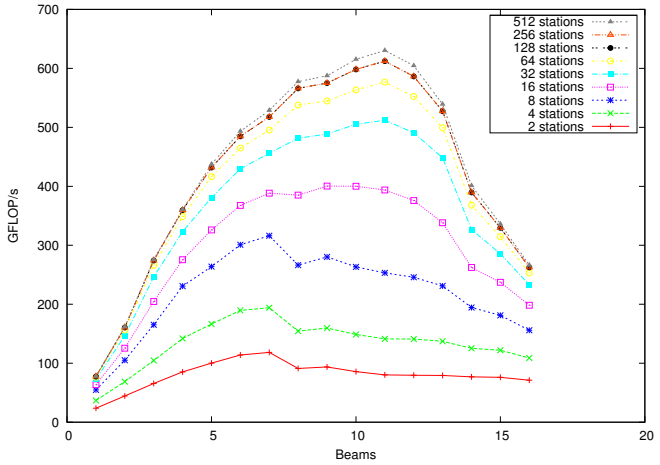

Figure 10. Tuning the beams-block for the NVIDIA GTX580 using CUDA (higher is better).

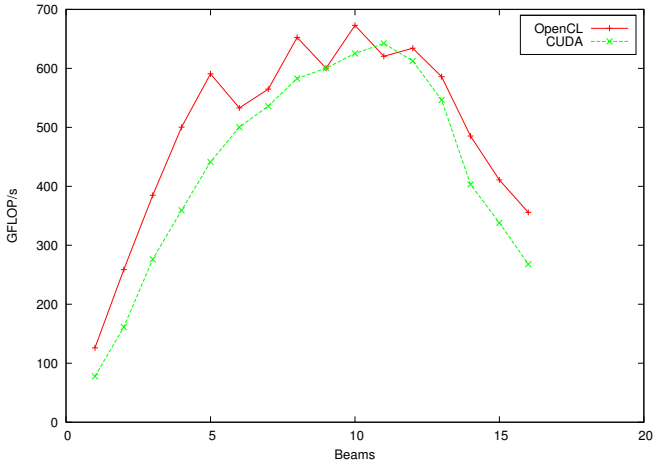

Figure 12. Comparison between CUDA and OpenCL on the NVIDIA GTX580, 512 stations (higher is better).

and beams because the number of stations is limited only by the available global memory (3 GB in our configuration). The number of beams, however, is limited by the number of available registers, and we hit the ceiling of 63 registers per thread (a hardware property of the NVIDIA Fermi architecture) before the value of 16 for the beams-block. The results using CUDA are presented in Figure 10.

The behavior appears to be regular, with performance increasing with the growing of the beams-block. We can see that for few stations, the performance's peak is found in correspondence of the value of 7, while for a large number of stations the same peak is found in correspondence of a beams-block of 11. We also show that for values larger than 11 beams, there is a sudden loss in performance, due to register spilling. The highest achieved number of GFLOP/s for the GPU beam former is 642, 40% of the card's theoretical peak performance. This is an excellent result for an algorithm that is as data intensive as the beam former.

The OpenCL implementation produces results that are comparable, as shown in Figure 11. We see, however, that the OpenCL behavior is more irregular: instead of smooth curves we have spiked ones. Differently from the CUDA implementation, the performance's peak appears to be in correspondence with a greater value of the beams-block for a little amount of stations, and to retreat back to the value of 10 for a high number of stations. Also the decrease in performance with large beams-block sizes is less steep.

We present the difference between OpenCL and CUDA with 512 stations in Figure 12. Overall, the OpenCL implementation provides slightly higher performance, reaching a peak of 672 GFLOP/s (42% of the theoretical peak), but the difference with the CUDA implementation is less than 2%. It is more difficult to tune the beams-block for the NVIDIA GTX580 than it was for the multi-core CPUs. In general

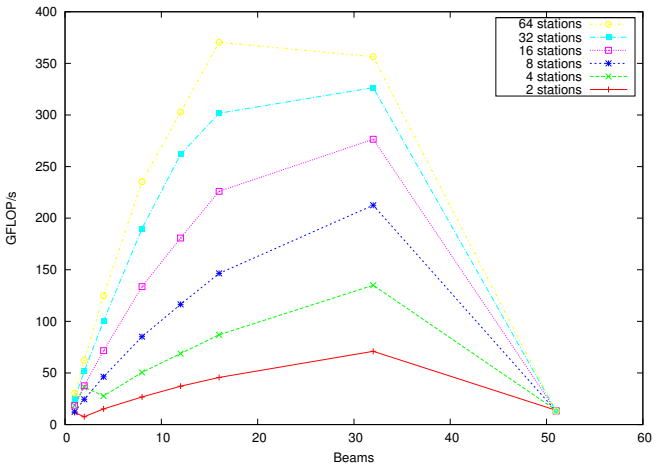

Figure 13. Tuning the beams-block for the AMD HD6970 using OpenCL (higher is better).

we can affirm that a value higher than 11 for the CUDA implementation, or 13 for the OpenCL implementation, badly affects performance. However, the best value of the beams-block is so dependent on the particular input size that a decision should be deferred until this value is known. When known, it can be easily computed thanks to the results of our auto-tuning step. More than for multi-core CPUs the auto-tuning looks of capital importance to achieve good performance with GPUs, as we can see that the performance window of these architectures is small.

E. AMD HD6970

The second GPU platform is the AMD HD6970, on which we run the OpenCL implementation of the beam former. For this platform we use different values for stations and beams: the number of stations varies between 2 and 64, and the number of beams between 1 and 51. These numbers reflect two large differences between this card and the NVIDIA GTX580: first, the HD6970 imposes limits on the amount of memory that can be allocated in a single block, thus reducing the maximum number of stations that is possible to merge within a single run; second, the HD6970 provides more registers per thread, thus increasing the beams-block space. This is caused by a difference in GPU architecture: AMD GPUs have a set of registers per core, while on NVIDIA GPUs, the register file is shared by all cores in a streaming multiprocessor. The results are presented in Figure 13.

It is interesting to observe that peaks are not confined to the beginning of the space: there are peaks at as high as a beams-block of 32, thanks to the large number of registers of this platform. Also the behavior is more stable than on the NVIDIA GPU, and the general advice for this platform is to use a larger value for the beams-block. Unfortunately, the impossibility of merge more than 64 stations in a single kernel execution prevents this platform to achieve

Platform	GFLOP/s	Efficiency
IBM Blue Gene/P	10.8	79%
Intel Xeon E5620 (OpenMP/SSE)	45	29%
Intel Xeon E5620 (OpenCL)	65	42%
AMD Opteron 6172 (OpenMP/SSE)	123	15%
AMD Opteron 6172 (OpenCL)	161	20%
AMD HD6970 (OpenCL)	370	13%
NVIDIA GTX580 (CUDA)	642	40%
NVIDIA GTX580 (OpenCL)	672	42%

Table II
MAXIMUM SINGLE RUN PERFORMANCE ACHIEVED DURING THE AUTO-TUNING.

performance comparable with the GTX580: we measure a peak performance of 370 GFLOP/s, only the 13% of the GPU's theoretical peak.

To conclude, Table II presents a summary of the highest measured GFLOP/s for all the platforms, together with the achieved efficiency. We see that our GPU beam former, compared with the production implementation, provides a performance improvement, per chip, of more than 60 times, even though the efficiency is lower due to the wide gap that GPUs have between memory bandwidth and computational power.

VII. Performance Analysis

In this Section we describe the performance and power efficiency results obtained by our many-core beam former, and compare the results of the different architectures with the production implementation.

A. Performance comparison for a sky survey observation

To answer the question if many-core architectures are suitable to accelerate beam forming in radio astronomy, we compare the performance of our beam former running on all the architectures presented in Table I with the production one, using a real use case from the LOFAR telescope.

For this experiment, the number of input stations is 64, the highest currently available in the LOFAR operational setup, with each station providing one second of observation, divided into 768 dual polarized samples, covering a frequency spectrum composed of 256 different channels. The number of beams to form as output is 155. This configuration is typical for a *sky survey observation*. The beams-block is set according to the results of Section VI. The achieved GFLOP/s, and the speedups relative to the Blue Gene/P implementation, are available in columns 3 and 4 of Table III.

In terms of raw computational power the GPUs are clear winners: our many-core beam former running on GPUs achieves 5–14 times more single precision floating point operations per second than on multi-core CPUs. The best performing platform is the AMD HD6970 GPU, achieving 612 GFLOP/s, 7% more than the same OpenCL implementation running on the NVIDIA GTX580. For what concerns the GTX580, we measure a difference of nearly 4% between

Platform	AI	GFLOP/s	Speedup	Watt	GFLOP/Watt
IBM Blue Gene/P	-	12.1	1	24	0.45
Intel Xeon E5620 (OpenMP/SSE)	1.92	42	3.4	360	0.11
Intel Xeon E5620 (OpenCL)	1.92	49	4.0	342	0.14
AMD Opteron 6172 (OpenMP/SSE)	1.92	106	8.7	625	0.16
AMD Opteron 6172 (OpenCL)	1.92	88	7.3	535	0.16
AMD HD6970 (OpenCL)	1.73	612	50.6	439	1.39
NVIDIA GTX580 (CUDA)	1.65	552	45.6	467	1.18
NVIDIA GTX580 (OpenCL)	1.63	572	47.3	455	1.25

Table III
PLATFORM COMPARISON FOR A TYPICAL SKY SURVEY OBSERVATION, MERGING 64 STATIONS INTO 155 BEAMS.

the CUDA and OpenCL implementations, with the latter performing slightly better than the former. It is important to note that, thanks to auto-tuning, our beam former shows performance portability between the different GPUs.

For what concerns the two multi-core CPUs, the OpenCL implementation provides higher performance than OpenMP/SSE on the Intel CPU, while on the AMD CPU it is the OpenMP/SSE implementation that performs better.

The IBM Blue Gene/P achieves 12.1 GFLOP/s in this configuration, 88% of the theoretical peak. In terms of efficiency this platform remains the best performing one, but, due to its design, it scores the lowest in terms of achieved GFLOP/s. Comparing the results of our many-core beam former with the production implementation, we measure an improvement of 3–8 times for multi-core CPUs and of 45–50 times for GPUs.

B. Power efficiency for a sky survey observation

Our GPU beam former provides high performance, as we demonstrated in Section VII-A, and can be tuned to different platforms and observation modes, as shown in Section VI. However, this is not enough: future software telescopes require high performance, but must also be highly power efficient. For LOFAR, power dissipation already accounts for a large part of the instrument's operational costs. For future telescopes, this will likely be even worse. When evaluating the computational architectures of future telescopes, it is thus necessary to look for an architecture that will maximize the number of operations that is possible to provide *per Watt of consumed power*. Therefore, we measure the power efficiency of our beam former for the different platforms we evaluate, again comparing with the production version on the Blue Gene/P. To measure power consumption, we run the different beam formers in the same operational environment (the sky survey observation) as described in Section VII-A, and use the DAS-4 Schleifenbauer Power Distribution Units (PDUs) to measure the power dissipated by the working nodes. The measurements are provided in Table III.

The IBM Blue Gene/P uses the least power per chip in absolute sense, as expected, since it is designed for low power consumption. However, the architecture also (by design) has a relatively low floating point performance per chip: it provides nearly half a GFLOP for each consumed Watt of power. This is not surprising, as the chip is created with an older manufacturing process (90 nm). Moreover, the Blue Gene/P also contains hardware for five different networks. These results are better than those achieved by the multi-core CPUs that we tested: they provide only 0.11–0.16 GFLOP per Watt. In contrast to the Blue Gene/P, however, these CPUs are focused on single core performance and not on power efficiency.

With 1.18–1.39 GFLOP per Watt, the GPUs are 2–3 times more power efficient than the Blue Gene/P, and 7–12 times more efficient than the multi-core CPUs we tested. Moreover, the GPU power efficiency increases when using more GPUs per node, since the power budget of the host is spread over more GPUs and more FLOPs. In fact, we experimented with a special DAS-4 node containing 8 NVIDIA GTX580 GPUs, resulting in a GFLOP per Watt ratio of 3.72. This is 8 times more efficient than the Blue Gene/P, and 23–33 times more than the multi-core CPUs.

VIII. CONCLUSIONS

Radio telescopes are quickly changing into gigantic sensor networks with complex real-time software pipelines. To efficiently implement future telescopes with exascale performance demands, we need to evaluate computing platforms that can provide high performance, while simultaneously being highly energy efficient. In this paper, we evaluated the beam forming algorithm, an important radio telescope building block, but also used in computer networks, radar systems and medical equipment. We evaluated this algorithm on seven many-core hardware and software platform combinations, while comparing to the production version of the beam former of LOFAR, the largest radio telescope in the world, which uses a Blue Gene/P supercomputer.

Optimizing memory access is of capital importance when dealing with platforms where the gap between computational power and memory bandwidth is wide, as it is on GPUs. This problem becomes increasingly important with virtually all modern architectures, as the number of compute cores grows much faster than the memory bandwidth. This is especially difficult with data-intensive algorithms, such as the beam former, which has a low arithmetic intensity.

We parallelized the algorithm for both multi-core CPUs and modern many-core GPUs. We implemented our solutions using OpenMP with SSE vector instructions, CUDA and OpenCL. To achieve high performance on these architectures, we modified the sequential algorithm and implemented many optimization techniques aimed at minimizing memory accesses. We maximized data reuse at different levels, both inter and intra-thread, while at the same time ensuring that the algorithm uses coalesced access to memory.

Since many-core platforms are changing rapidly, and telescopes have lifetimes of decades, we do not focus on a specific many-core architecture, but aim to be portable, in terms of both code and performance. We demonstrated that it is possible to use auto-tuning to achieve high performance for different combinations of hardware platforms and implementation frameworks. We use run-time compilation techniques to automatically tune the code for a particular input problem (observation specification) and hardware platform dynamically. We showed that performance portability is possible in practice: our auto-tuned OpenCL code achieves the same or better performance than hand-optimized code, on both GPUs and multi-core CPUs. Moreover, we believe that this approach to code and performance portability, based on run-time code generation and auto-tuning, may be applied to different parallel applications for many-core architectures. We are now applying this same approach to other radio astronomy algorithms.

Our approach leads to exciting results: compared to the production implementation, our auto-tuned beam former is 45–50 times faster and 2–3 times more power efficient on GPUs, and even 8 times more power efficient when using 8 GPUs in a single node. Furthermore, the GPU solution remains the fastest even when taking the host-GPU memory transfers into account. We conclude that GPUs provide a viable solution for high performance and energy efficient beam forming in radio astronomy. We plan to further investigate how GPUs may be used to accelerate other software telescope pipeline components. In addition, we want to study the feasibility of performing the entire LOFAR real time pipeline on a GPU-powered cluster.

References

[1] A. de Bruyn *et al.*, "Exploring the universe with the low frequency array, a scientific case," September 2002, http://www.lofar.org/PDF/NL-CASE-1.0.pdf.

[2] NVIDIA Corporation, "NVIDIA CUDA C programming guide 4.0," May 2011.

[3] Khronos OpenCL Working Group, "The OpenCL specification 1.1," September 2010.

[4] J. D. Mol and J. W. Romein, "The LOFAR Beam Former: Implementation and Performance Analysis," in *EuroPar'11*, vol. LNCS 6853, no. Part II, Bordeaux, France, August 2011, pp. 328–339.

[5] J. W. Romein, P. C. Broekema, J. D. Mol, and Rob V. van Nieuwpoort, "The LOFAR correlator: Implementation and performance analysis," in *15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP 2010)*, Bangalore, India, January 2010, pp. 169–178.

[6] J. Roy, Y. Gupta, U. Pen, J. Peterson, S. Kudale, and J. Kodilkar, "A real-time software backend for the GMRT," *Experimental Astronomy*, vol. 28, pp. 25–60, 2010.

[7] B. Mort, F. Dulwich, S. Salvini, K. Adami, and M. Jones, "OSKAR: Simulating digital beamforming for the SKA aperture array," in *IEEE International Symp. on Phased Array Systems and Technology (ARRAY)*, oct. 2010, pp. 690–694.

[8] C. Nilsen and I. Hafizovic, "Digital beamforming using a GPU," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2009, pp. 609–612.

[9] X. Lian, H. Nikookar, and L. Ligthart, "Efficient radio transmission with adaptive and distributed beamforming for intelligent WiMAX," *Wireless Personal Communications*, vol. 59, pp. 405–431, 2011.

[10] T. Helzel and M. Kniephoff, "Software beam forming for ocean radar WERA features and accuracy," in *IEEE OCEANS 2010*, 2010, pp. 1–3.

[11] D. Byrne, M. O'Halloran, M. Glavin, and E. Jones, "Contrast enhanced beamforming for breast cancer detection," *Progress In Electromagnetics Research*, vol. 28, pp. 219–234, 2011.

[12] OpenMP Architecture Review Board, "OpenMP Application Program Interface 3.1," July 2011.

[13] S. Thakkur and T. Huff, "Internet streaming SIMD extensions," *IEEE Computer*, vol. 32, no. 12, pp. 26–34, 1999.

[14] A. Sclocco, "Radio astronomy beam forming on GPUs," May 2011. [Online]. Available: http://alessio.sclocco.eu/wp-content/uploads/MScThesis.pdf

[15] J. Meredith, P. Roth, K. Spafford, and J. Vetter, "Performance implications of non-uniform device topologies in scalable heterogeneous GPU systems," *IEEE Micro*, vol. PP, no. 99, p. 1, 2011.

[16] Rob V. van Nieuwpoort and J. W. Romein, "Correlating radio astronomy signals with many-core hardware," *Springer International Journal of Parallel Programming*, vol. 39, no. 1, pp. 88–114, 2011.

[17] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in *13th ACM SIGPLAN Symp. on Principles and practice of parallel programming (PPoPP)*, 2008, pp. 73–82.

[18] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Commun. ACM*, vol. 52, pp. 65–76, April 2009.

[19] "Das-4: The distributed ASCI supercomputer version 4," see http://www.cs.vu.nl/das4/.