VRIJE UNIVERSITEIT AMSTERDAM

Auto-tuning a LOFAR radio astronomy pipeline in JavaCL

Author
Jan Kis

Supervisors
Rob V. van Nieuwpoort Ana Lucia Varbanescu

netherlands eScience center

VU VRIJE UNIVERSITEIT AMSTERDAM

A thesis submitted in fulfillment of
the requirements for the degree of
Master of Science

in the
Faculty of Sciences
Department of Computer Science

Amsterdam 2013

I would like to express my deepest gratitude to Ana Lucia Varbanescu and Rob V. van Nieuwpoort for their professional supervision and sincere interest in making this a better work. I should never forget our long discussions over a cup of tea and "bitterkoekjes" after which I felt both academically enlightened and high-spirited.

Abstract

Modern radio telescopes, such as the Low Frequency Array (LOFAR) in the north of the Netherlands, process the signal from the sky in software rather than expensive special purpose hardware, This gives the astronomers an unprecedented flexibility to perform a vast amount of various scientific experiments. However, designing the actual software that would give optimal performance for many different experiments, possibly also running on different hardware is a challenging task. Since optimizing the software by hand to fit the various experiments and hardware is unfeasible, we employ a technique called parameter auto-tuning to find the optimal solution. Auto-tuning is based on a construction of a more generic software which has the ability to explore its parameter space and choose the values optimal for the experiment and hardware at hand. In our work we devise a systematic approach for auto-tuning signal processing pipelines which is not limited to radio astronomy and can be easily extended to other fields as well. Since the computations in radio astronomy are suitable candidates for parallelization, we exploit the power of GPUs. For that purpose we use the OpenCL programming standard. Furthermore, we combine OpenCL with Java to gain the advantages of higher level programming language such as greater productivity and code portability.

Contents

Chapter 1

Introduction

The broader aim of this project is to help astronomers in their day to day work when observing the sky and trying to answer the countless questions about our universe. Within the wide field of astronomy we focus ourselves on radio astronomy which studies celestial objects through radio waves they emit. Unlike the traditional, optical astronomy the radio astronomy is interested in wavelengths greater than visible light, and can therefore detect and examine broad range of celestial objects not visible by optical telescope (such as supernovas and pulsars).

The first radio telescopes emerged in the late 30s. Consisting of only a single antenna they were quite simplistic and often experiment specific. Nowadays, the radio telescopes consist of large arrays of antennas which form a single virtual gigantic telescope. This allows the astronomers to perform various kinds of observations and experiments. For instance, astronomers can choose a celestial object of their interest based on the frequency range the object emits. Additionally, one can also choose the direction, area and resolution of the observation.

The telescope connected to our work is called Low Frequency Array (LOFAR) [28]. It was constructed and is used by the Netherlands Institute for Radio Astronomy (ASTRON). In contrast to other radio telescopes, which use large and expensive satellite dishes to receive signal from the sky, LOFAR consists of many simple omni directional antennas (Figure 1.1). We refer to a group of these antennas placed closely together as a station. Currently, LOFAR has around 40 stations, most of them in the north of the Netherlands. However, there are few stations situated outside of the Netherlands in Germany, U.K., France and Sweden [2].

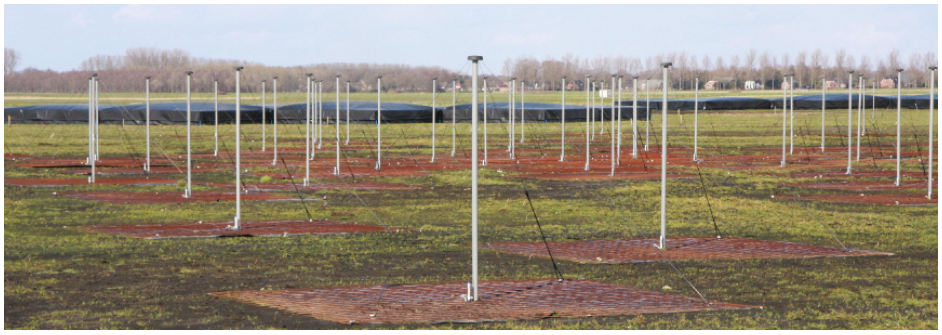


Figure 1.1: LOFAR antennas (single station).

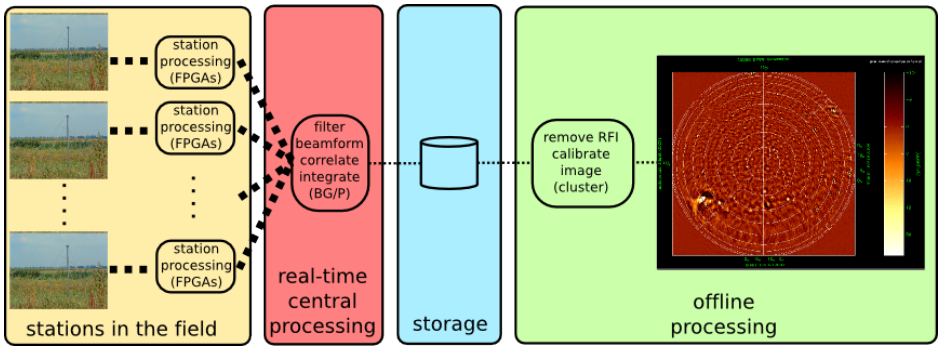


Figure 1.2: A simplified overview of LOFAR processing

LOFAR, unlike other radio telescopes, uses a novel approach of processing the signal coming from different stations in software rather than in expensive, custom made hardware. A simplified overview of the LOFAR signal processing is depicted in Figure 1.2. Only the first part (within a single station) happens in hardware using FPGAs. The rest of the processing is done in software.

The crucial part of the LOFAR processing happens in real-time (pink/red box in Figure 1.2). This is due to the enormous amount of data the stations receive. Since it is not possible to store all the data, we need to reduce it in real-time and store it afterwards. In our work, we concentrate entirely on the real-time processing which is fairly dynamic and can be configured according to the kind of observation astronomer wants to perform. The real-time processing consists of several smaller computational parts called *kernels*. Usually a single astronomical observation requires a connection of multiple kernels referred to as *pipeline*. Different observations might require connection of different kernels (pipelines). For example, if we were interested in observing a certain area of the sky we would use a pipeline of a band-pass filter followed by a beam former kernel.

Our main goal is to perform increasingly large observations. Ideally, we would like to perform even bigger observations, than LOFAR currently supports, to lay down a path for an important international project called the Square Kilometer Array (SKA) [8]. Since SKA intends to be the largest and most sensitive radio telescope ever built, we need to scale up in terms of:

- the number of stations observing the sky
- the frequency range of the signal
- the explored area of the sky during a single observation

Computation-wise performing larger observations means to process larger amounts of data in the same amount of time which in turn stresses the importance of having the LOFAR pipeline processing as fast as possible.

To achieve the best possible performance, currently, LOFAR uses the Blue Gene supercomputer from IBM. However, using Blue Gene has several disadvantages:

- it is expensive to run and maintain
- it gets outdated and needs to be replaced every 4 years
- it is vendor specific

In our work we exploit many-core architectures, especially GPUs as viable and cheaper alternatives to Blue Gene. To avoid limitation to single hardware we use OpenCL language which allows our solution to run on broad range of architectures ranging from multi-core CPUs through Cell Broadband Engine to GPUs.

A further important aspect of our effort to achieve the best possible performance is a technique called auto-tuning. It allows us to explore different parameters of our solution and pick the values most suitable for the underlying hardware. Furthermore, different kinds of observations might have different optimal parameter values and auto-tuning helps us to find them.

In this context the main question we address in this work is: **How much performance can we gain by auto-tuning the LOFAR pipeline?** We split this question into three more generic research questions:

- **How to auto-tune a single kernel?**
  First of all, we need to know how to optimize each kernel in the pipeline. The answer to this question will therefore be a systematic method for creating a kernel suitable for auto-tuning.

- **How to efficiently connect kernels?**
  Once we have optimal kernels, we need to find out how do we connect two kernels. This can be a challenging task; especially, when the two optimal kernels use data in different formats. We will explore two generic approaches for connecting any two kernels.

- **How to auto-tune an entire pipeline?**
  Finally, when we know how to connect the kernels, we can explore how do we make them perform optimally together within a pipeline. The answer should generalize from the 2 kernels problem to the N kernels as well as evaluate the performance of the entire pipeline.

The structure of the thesis:
In the remainder of the thesis we first introduce the most important LOFAR kernels and closer explain the advantages of auto-tuning (Chapter 2). In Chapter 3 we present the used programming model of our solution. We take a look at related work in Chapter 4. We address our first research question (How to tune a single kernel?) in Chapter 5. Then, we examine how to connect two kernels in Chapter 6. Finally, in Chapter 7 we tune an entire pipeline of kernels. We conclude by summarizing our most generic findings and presenting new ideas for future work (Chapter 8).

Chapter 2

Background

In this chapter we first present necessary information on the LOFAR pipeline (Section 2.1). Afterwards, we explain the term auto-tuning and the motivation behind it (Section 2.2).

2.1 LOFAR pipeline

The most important kernels from the real-time LOFAR pipeline are the poly-phase filter (PPF), the beam former and the correlator. When talking about the *LOFAR pipeline* we mean the connection of the LOFAR kernels (PPF, beam former and correlator). When mentioning a *generic pipeline*, we refr to a connection of several arbitrary kernels. An introduction to the PPF, beam former and correlator follows.

2.1.1 Poly-phase filter

The poly-phase filter is the first step in the LOFAR pipeline. Its purpose is twofold. First, the signal is filtered by the Finite Impulse Response (FIR) filter which adjusts each sample according to $N - 1$ previous samples. Each of the $N - 1$ previous samples and also the filtered sample is multiplied by a real coefficient (weight) and the results are summed [38]. The number of the combined

samples (N) is referred to as the *number of taps*. Secondly, the signal is split into more frequency channels at the cost of lower time resolution by the Fast Fourier Transformation (FFT).

Since it is out of the scope of this work, we did not implement our own FFT. Instead, we used an FFT implementation already available for the language of our choice (JavaCL). The disadvantage of the JavaCL FFT is that it is roughly 100 times slower than the fast CUFFT library from NVIDIA [25]. As the development efforts for an optimized FFT are too large we settle for this solution. However, for a fast production pipeline, further effort needs to be invested in writing an OpenCL/JavaCL FFT to get close to the performance of CUFFT.

Further on we often treat the FIR filter and FFT as two separate kernels.

2.1.2 Beam former

The beam former step of the LOFAR pipeline allows the astronomers to direct their observations to a certain area of the sky. It combines the signal received by different stations to create one big virtual directional antenna.

However, a straight forward signal combination without any modification is not enough since the antennas receive the same signal with different phase shifts due to the difference in their spatial positions. As a result, before combinig the signals we need to adjust them. The signal adjustment is done by simply multiplying each signal with a weight based on station's position and observed source location. Afterwards, a beam is formed by summing up the signals from all stations [32].

Usually we calculate several beams. To cache the access to the slow global memory and reuse the signal of a single station, when calculating different beams, we compute a block of beams at once. The whole block of beams is loaded into registers so that when we iterate through stations we can reuse a single station for each beam within the beam block.

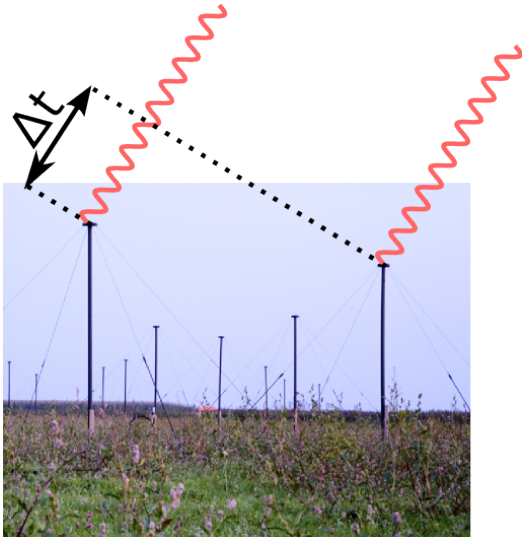


Figure 2.1: Signal phase shift due to receive time delay.

2.1.3 Correlator

The correlator kernel extracts the relevant signal from the data coming from the stations. It removes the noise by correlating signals from all pairs of stations [39, 28]. Additionally, it can also reduce the data size (when the number of observations per second is larger than number of stations squared, which usually is the case).

From computational aspect, correlating signals from two stations means multiplying the signal of the first station with a complex conjugate of the other station signal. We call a correlated pair of two stations a baseline (Figure 2.2). When calculating the baseline for a pair of stations (a, b) it is enough to calculate just one pair. In other words, the baseline of (b, a) can be easily deduced from (a, b); hence, the triangle instead of a square in Figure 5.3. Similarly as with the beam former, to exploit the speed of registers, we group the stations in cells. We load all stations within a single cell into registers and calculate all the baselines formed by the loaded stations; thus, we reuse the stations and reduce the access to the global memory.

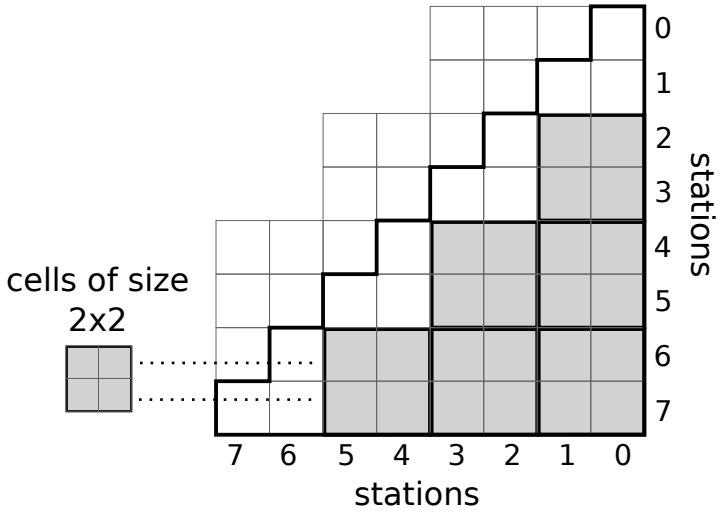


Figure 2.2: Triangel with calculated baselines divided to cells

2.2 Auto-tuning

To explain auto-tuning, we need to define manual tuning. Manual tuning, in our perception, means exploring the values of different application parameters by hand. For example, if we have a numeric parameter, we perform several runs with different values and choose the best performing value. We can also have more complex parameters than just numeric ones, which can require the construction of multiple versions of a kernel, and running each kernel to find the best performing one. For instance, the correlator's cell size and shape is such a parameter (Section 5.2.1).

In contrast to manual tuning, the automatic tuning (auto-tuning) is a mechanism operating upon the application which searches through parameter values (ideally all possible values) automatically and chooses the best one. There are multiple advantages to auto-tuning:

- Auto-tuning brings performance portability. We can tune an application on any hardware it is intended to run at, and thus obtain the optimal parameter values for many different architectures.

- Auto-tuning can easily explore large ranges of values which would be tedious or even impossible to explore by hand.
- In the context of our work, auto-tuning allows us to find optimal parameter values for different astronomical observations. The observations can differ for example in number of used stations or in the kernels connected in the pipeline.

The trade-off for performing auto-tuning is the need to implement a generic kernel which is capable of exploring all parameters' values. Nevertheless, we still believe that implementing auto-tuning is faster than manual tuning and yields more generic, better extendable code.

Chapter 3

Programming model

LOFAR currently runs its pipeline on the Blue Gene supercomputer. Blue Gene is very expensive, vendor specific and compared to GPUs also power consuming. Therefore, we experiment with parallelizing the kernels on many-core architectures, mainly GPUs. To utilize the power of GPUs, we use technology called OpenCL (Section 3.1). We combine OpenCL with Java, through a library called JavaCL (Section 3.2), mainly because of the easier development process. Finally, in Section 3.3 we compare the performance of JavaCL to OpenCL and CUDA to see whether we are loosing any performance due to our choice of Java.

3.1 OpenCL

To get the best performance out of our pipeline, we mainly explore the parallelization of the pipeline on GPUs. To achieve code portability and be able to run the pipeline on several different GPUs or even multi-core CPUs, we use OpenCL. OpenCL is a general purpose programming standard, implemented by different hardware vendors as a C library, suitable especially for high performance parallel computing [7].

OpenCL gives us a code portability across a broad range of architectures. When not choosing OpenCL, we might need at least one implementation for CPUs, one for NVIDIA GPUs and another one for ATI GPUs.

The programming model of OpenCL language closely maps the hierarchical computing and memory models of GPUs [23]. In OpenCL one writes a single piece of code (kernel) which is run by a large number of threads (work items). All the work items can access the GPU's global memory. Furthermore, the work items are grouped in work groups where the items from a single group can access a shared memory (in OpenCL called local memory). The shared memory is often used as cache as it is faster than the global memory.

OpenCL distinguishes between the code run on device (usually GPU) and the code run on host (usually CPU). Nevertheless, if there is no GPU the device and host usually refer to the same CPU.

3.2 JavaCL

For our implementation we decided to combine OpenCL with Java to get the advantages of a higher level programming language. Although C or C++ would be a more natural choice since OpenCL is a C library, we chose Java for its simplicity and increased productivity. What is more, this enables our future plans to distribute the computation across several hosts with the IBIS/IPL Java library [3].

There are multiple Java/OpenCL libraries that wrap the OpenCL methods and allow the programmer to call OpenCL from Java. First, we looked at open source library supported by AMD called Aparapi [1]. In Aparapi the programmer writes the OpenCL kernels in Java by extending a simple *Kernel* interface. This approach not only very nicely supports Java's Object model but it also allows transparent (to the programmer) data copying to and from the GPU. On the minus side of Aparapi is the fact that the programmer looses the fine control over the kernel and can not use some aspects of OpenCL such as shared memory for example. Another library we considered is called JOCL [5]. However, the outdated documentation and failure to install it rendered this library useless. Finally, we examined and decided to use JavaCL open-source library [4]. JavaCL incorporates the OpenCL nicely into the Java Object model. It allows calling the kernels written in plain OpenCL, and thus the programmer does not loose the fine control in contrast to Aparapi. What is more, it has a solid user base and active forum (unlike JOCL).

Optimization	CUDA	OpenCL
cells (registers)	yes	yes
shared memory	yes	no
vector operations	n.a.	yes

Table 3.1: Comparison of the optimizations present in the reference implementations (n.a. stands for not available)

3.3 JavaCL vs OpenCL vs CUDA

In this section we verify how does JavaCL perform in comparison with plain OpenCL and CUDA. CUDA is a language specific for NVIDIA GPUs with a programming model very similar to OpenCL. All the experiments are performed on the correlator kernel. Since our computation runs almost entirely on device, we compare only the kernel execution times.

Our JavaCL implementation originated from two reference implementations: One written in CUDA and used in [39] and another one written in OpenCL. The comparison of the different optimizations available in the two implementations is presented in Table 3.1. The CUDA implementation achieves better performance than the OpenCL mainly because of the possibility of using the shared memory. Both versions can calculate cells of fixed size only.

Having the two reference implementations, we developed the JavaCL version of the correlator in two steps. First of all, we ported the existing OpenCL version to JavaCL. This was done by rewriting the existent C host code into Java. Since both implementations (C and Java) use the same kernel and also the same OpenCL compiler, we expect the performance to be almost identical. The performance comparison presented in Figure 3.1 proves that JavaCL imposes no overhead over OpenCL.

Secondly, inspired by the CUDA reference implementation, we optimized the kernel by adding shared memory. To verify how the JavaCL optimized kernel competes with CUDA, we compared the two versions in Figure 3.2. The comparison shows that our optimized JavaCL version can even slightly outperform the CUDA reference implementation.

All in all, we confirmed that our choice of calling OpenCL from Java is no slower than using plain OpenCL. As a result, we can enjoy the ease of programming in a higher level language (Java).

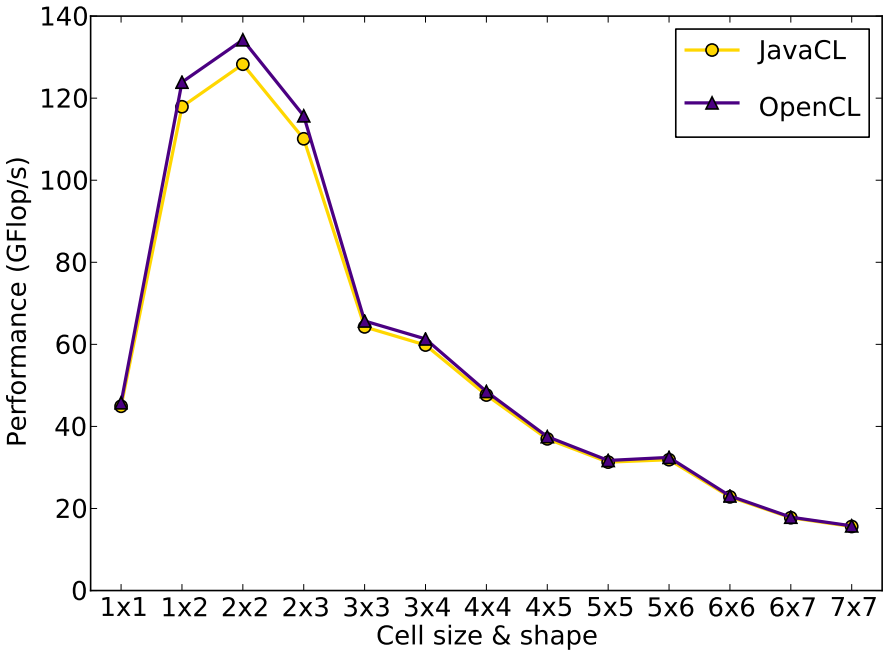
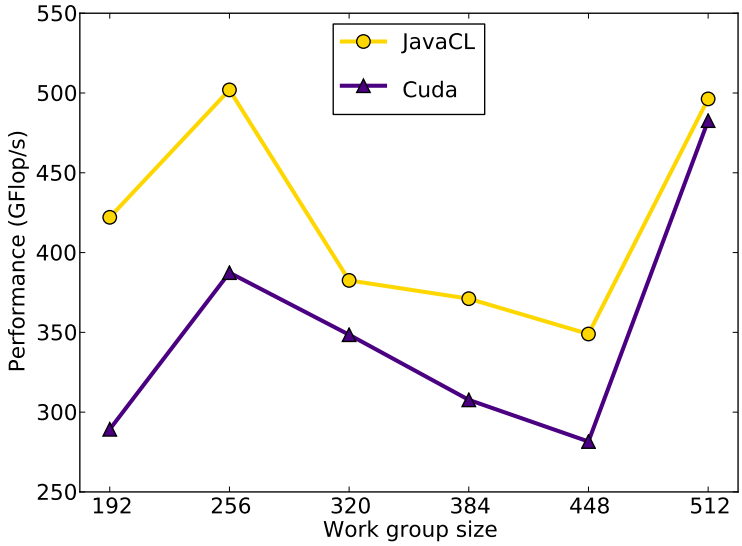
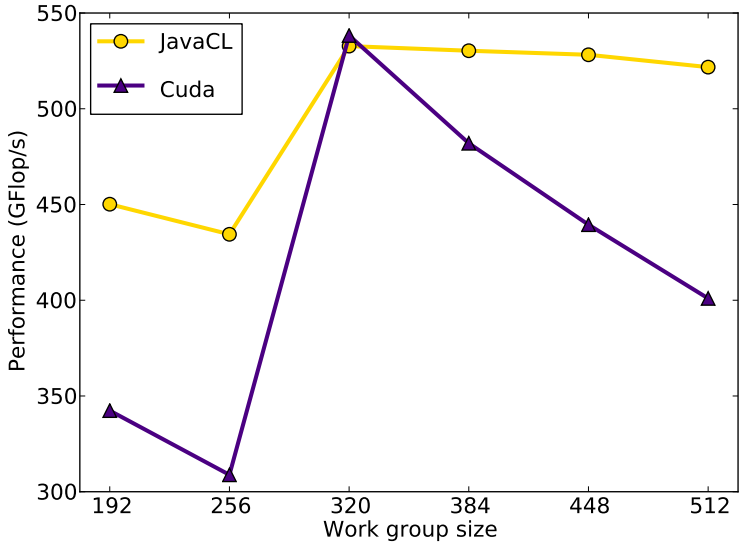


Figure 3.1: JavaCL vs OpenCL comparison on GTX 480 without shared memory (higher is better)



(a) Cell size 1x4



(b) Cell size 1x6

Figure 3.2: JavaCL vs CUDA comparison GTX 480 with shared memory (higher is better)

Chapter 4

Related work

In this chapter we discuss the related work which we divide according to the three topics we touch upon in our work. First, we discuss the work done on radio astronomy signal processing (Section 4.1). The core of the chapter is devoted to auto-tuning (Section 4.2). We finish the chapter presenting previous work on data transformation (Section 4.3).

4.1 Radio astronomy pipelines

Up until recently, radio telescopes processed the received signal in hardware. This solution requires special purpose hardware to be designed. Producing it is not only extremely costly but also time consuming. Furthermore, such hardware is rarely capable of adjusting to the different kinds of observations the astronomers would like to make.

As a consequence, software radio telescopes emerged. To our knowledge, the first such a telescope was the *Low Frequency Array* (LOFAR) [16, 17] from the Netherlands Institute for Radio Astronomy (ASTRON). LOFAR is a pathfinder for an ambitious project to construct the largest radio telescope *Square Kilometer Array* (SKA) [8] which should serve to find answers to fundamental questions about the origin of universe. Other SKA pathfinders are the MeerKat and Askap telescopes stationed in south Africa and Australia respectively [15, 6]. After LOFAR, other radio telescopes started to migrate from hardware to software solutions. For

example, the Giant Metrewave Radio Telescope (GMRT) [29]. The U.S. National Radio Astronomy Observatory (NRAO) is also researching the potential of software solutions [18]. The results of these instruments prove that using software in modern radio telescopes is a viable solution.

In our work we concentrate on the LOFAR real-time software pipeline [28]. We are specifically interested in using many-core hardware architectures which should replace the currently used Blue Gene super-computer. The previous work on the specific kernels (poly-phase filter [38], beam former [32] and correlator [39]) within the LOFAR pipeline already suggest that many-core hardware and especially GPUs have the potential to replace Blue Gene. Our work further extends the studies on poly-phase filter, beam former and correlator. Exploiting auto-tuning we perform a thorough analysis on the performance which can be gained by connectint the kernel is one pipeline.

4.2 Auto-tuning

The need for auto-tuning and its importance in high performance computing have been advocated in several studies, for example in analysis on parallel computing from Berkeley [11]. We characterize these studies according to different criteria:

- **Time of tuning**
  Auto-tuning can happen either online or offline. Offline tuning happens before the actual run in which we want to achieve the best possible performance. Offline tuning is appropriate when we want to perform a lot of experiments with the same setup on the same hardware. Most of the existing work focuses on offline tuners. Online (or run-time) tuning optimizes the application during each run. As [10] suggests, this type of auto-tuning is suitable for cloud computing services with heterogeneous environments. A different, but still interesting idea is incorporating the auto-tuning into the operating system and tune every running application at the OS level [36].

- **Object of tuning**
  Many studies in auto-tuning deal with compiler tuning [9, 37]. This kind of work mainly addresses the exploration of various *loop* transformations for C and Fortran compilers. At the opposite side of the spectrum are what we call the *application tuners*. These have broader scope and explore the problem, the implementation and also the technology parameters rather than focusing solely on low level *loop* optimizations (for example [27, 41]).

- **Tuning approach**
  Choosing the optimal parameters' values is not trivial since there are usually many possible combinations. Based on how does the tuner choose the optimal parameter values, we can divide them into model based and search based. The model based tuners try to avoid search by using a model closest to the underlying hardware [14, 43]. The search based tuners can be further divided by the method they use to explore the search space:

  - exhaustive search
  - hill climbing
  - random sampling
  - dynamic programming
  - evolutionary algorithms
  - machine learning

  From the search based tuners, the modst interesting is the SPIRAL framework [27] which implements and compares multiple of the above mentioned search techniques.

- **Tuner applicability**
  Based on the type of problems for which we can use the auto-tuner, we can distinguish three groups of tuners. The first group consists of problem specific tuners which can handle only one type of problem [20, 41, 14]. The second group includes domain specific tuners which can handle a range of problems from the same domain - for example, linear algebra matrix operations and stencil calculations [12, 40, 22]. Although being problem and domain specific, and not closely related to our work, these two groups can still have an inspirational value by showing successful auto-tuning methodologies. For instance, in [41] the authors introduce the Roofline model [42] to rank the optimizations to be applied and only choose those that have the potential to bring actual performance improvement. The third group represents generic tuning frameworks which can be used for arbitrary problems. We found the FLAMINGO auto-tuning framework [33] (actively employed in a broader project OP2) [21] especially interesting. FLAMINGO is written in Python and it tunes the application by exploring the values of application's command line arguments. It is generic enough to specify the metric which we want to maximize or minimize. Another generic tuning framework we briefly studied is called Atune-IL [31]. It is based on *pragmas* in the source code which identify the parameters to be tuned. In each run Atune-IL replaces the *pragmas* with specific parameter values that should be explored.

In the light of the above classification, we designed and build a custom made, offline, application tuner performing exhaustive search. Since the hardware for LOFAR rarely changes and there is only a limited number of different observations, which we need to tune for, an offline tuner is enough for us. Furthermore, for the ease of implementation we use the exhaustive search. In the end we were able to prune it significantly by hand. Finally, we decided to implement our custom tuner instead of using one of the generic tuners because of their lack of flexibility in generating sequences of parameter values (like integer sequences or permutations).

4.3 Data layout transformation

The topic of data reshuffling or data layout transformation is a fairly new topic in many-core programming. However, especially in the case of GPUs, layout transformation is of a key importance as the layout has critical performance impact.

The significance of the data layout in two dimensional matrices is introduced in [30] together with a solution for a fast out-of-place transposition. An API for data layout transposition called Dymaxion [13] takes it a bit further and offers transformations to several layouts. However, it still discusses two dimensional matrices only.

A considerable step forward has been taken at the University of Illinois in studies on general data lyout transformations. In [35] a data layout formalism capable of expressing any number of dimensions is introduced. The work not only considers a dimensions permutation as layout transformation but it also discusses dimension splitting. The optimal layout is chosen based on a suitable hardware model. In [34] transformation from any layout to a common, custom made layout called *array of structure of tiled arrays* (ASTA) is discussed. The work considers an in-place transposition based on finding cycles among new indices. Unfortunatelly, this approach can lead to a poor performance on GPUs due to the fairly random memory access pattern of single threads.

Our work uses a generic data layout which is similar to the formalism in [35]. We focus ourselves only on dimensions permutation and skip dimension splitting. Additionally, due to the low number of dimensions, we explore all possible layouts. In contrast to [34], we use a fast out-of-place transformation.

4.4 Summary

Overall, the previous studies presented in this chapter explore either the auto-tuning of a single kernel or data layout conversions or software radio astronomy pipelines but none of the studies brings these three topics together. Consequently, our work can be considered unique in employing the notion of auto-tuning (especially data layout tuning) within the context of an entire pipeline of kernels.

Chapter 5

Kernel auto-tuning

Although the OpenCL code (and also the JavaCL code) is portable across several architectures, its performance is not. A traditional approach is to hand-tune the kernel for every architecture the kernel is expected to run at. However, it would take considerable amount of time, it would require to study each of the hardware architecture in great detail and in the end the solution would be most probably not portable (performance-wise) to new architectures. On the other hand, one can use automatic kernel tuning (auto-tuning) to find the optimal set of parameters' values and optimizations for any architecture. We believe that auto-tuning is less costly in terms of development time and the knowledge required about all the potential hardware.

In this chapter we show a systematic approach for tuning a single kernel. We focus ourselves on the computationally most complex kernel of our pipeline, the correlator kernel. We first identify the parameters which we would like to tune (Section 5.1). Afterwards, in Section 5.2 we discuss the challenges of constructing a tunable kernel. Then, we perform the actual tuning in several experiments (Section 5.3). Finally, we conclude by stating our findings in Section 5.4.

5.1 Identification of parameters

In order to successfully employ auto-tuning one first needs to identify the parameters with the greatest performance impact. These parameters will be later on tuned. We identified and categorized the parameters as follows:

- **Technology specific**
  These parameters pertain to the language and technology used. In our case, these are OpenCL specific parameters:
  - Work group size
  - Number of work groups
- **Problem specific**
  These parameters are specific to the problem at hand. In our case, these are parameters characteristic for correlating radio astronomy signals:
  - Number of stations
  - Number of channels
  - Number of time samples

  Although, the number of stations and channels is specified by user (it is not something a programmer can choose freely), it does make sense to include them in the auto-tuning. For instance, it helps us to answer a question of how do the optimal parameter values differ for different number of stations. Furthermore, it can give indications on which numbers are favorable, offering choices for the astronomers in setting up their experiments.
- **Implementation specific**
  These parameters are specific to the particular implementation. In our case, these are parameters specific to our implementation of the correlator kernel:
  - Cell shape
  - Cell size
  - Use of shared memory

5.2 Construction of tunable kernel

After identifying the parameters, it is essential to make the kernel tunable by allowing exploration of each parameter value space. For example, in the case of the correlator kernel we need to be able to explore the space of cell sizes (Section 5.2.1). Ideally, we would like to explore the space in a continuous fashion (be able to explore every possible value). Once we are able to explore the different cell sizes we need *full tiling* (different cell division scheme) for large cells to improve the reliability of our measurements 5.2.2. Implementing these requirements means making the code flexible. This is often not trivial and time consuming.

5.2.1 Generic vs specific kernel

To auto-tune the correlator kernel we have to be able to run the kernel on cells of arbitrary size. To accommodate for cells of any width and height we implemented dynamic kernel generation. A specific kernel is generated for a given cell width and height.

Another way to calculate cells of any size is to have a single generic parametrized kernel that loads samples to array and iterates through it. Arrays are usually accessed in for loops which causes that they need to be stored in global memory rather than in registers. Consequently, one tries to avoid the use of small arrays which can possibly fit into registers. Nevertheless, we implemented a generic kernel and compared it to the generated specific kernel (see Figures 5.1, 5.2). Surprisingly, the performance of the generic kernel (using the arrays) was in most cases comparable to the specific kernel. A further inspection of the NVIDIA's pseudo assembly code (PTX) generated by the compiler revealed that the two kernels have the same amount of loads and stores from global memory. Judging from these observations we assume that the compiler is able to unroll the for loop accessing the array; hence, it can use registers instead of global memory. The only case when the generic kernel performs significantly slower than the specific one occurs when using cells of size N x 1 where N denotes cell width and has value greater than 2. The analysis of the PTX code shows lower register usage and higher number of loops. In another words, the NVIDIA compiler fails to unroll the loop and consequently also fails to allocate the array in registers.

Having proved that the generic kernel can compete with the specific kernel the further correlator experiments will use the generic kernel unless stated otherwise.

5.2.2 Full tiling

In this section we use word *tiling* to refer to the division of baselines to cells. In our solution we calculate only those baselines which are part of the tiling (i.e. they lie within a cell). We call the baselines which lie within a cell and also within the calculated triangle *regular baselines*. The baselines which lie within the calculated triangle but outside of the tiling we call *missing baselines*. Finally, the baselines which are outside of the calculated triangle we call *unwanted baselines*. The three kinds of baselines are illustrated in Figure 5.3. Note that when the tiling covers the entire triangle or exceeds it we have 0 *missing baselines*.

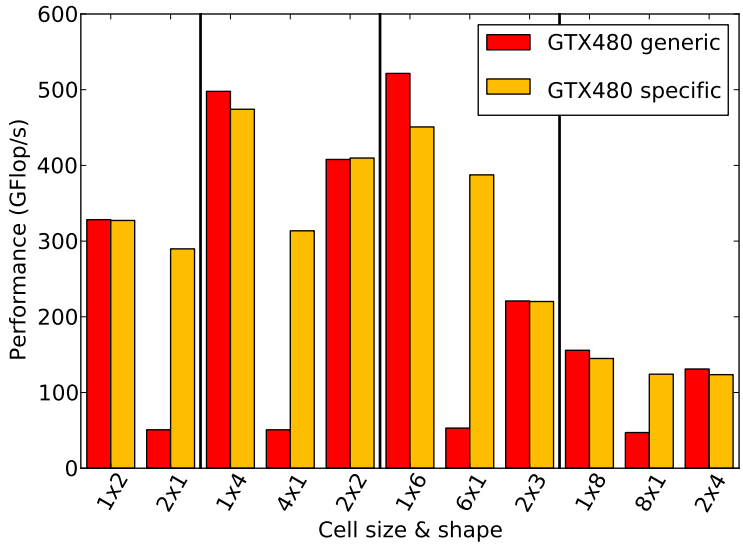


Figure 5.1: Generic vs specific kernel comparison on GTX 480 (higher is better)

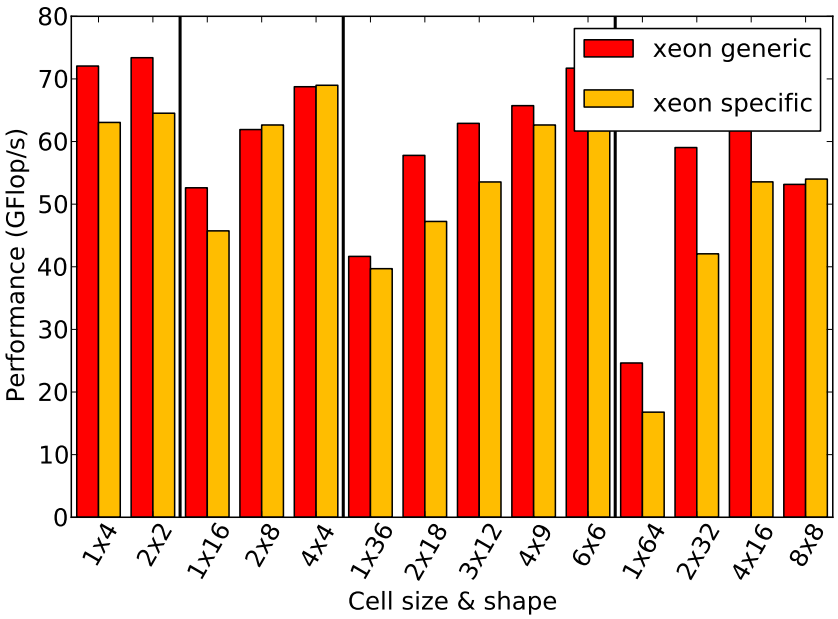


Figure 5.2: Generic vs specific kernel comparison on Intel Xeon (higher is better)

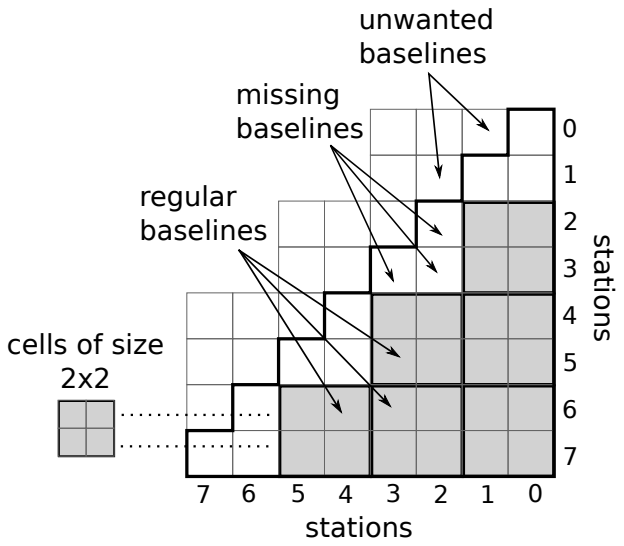
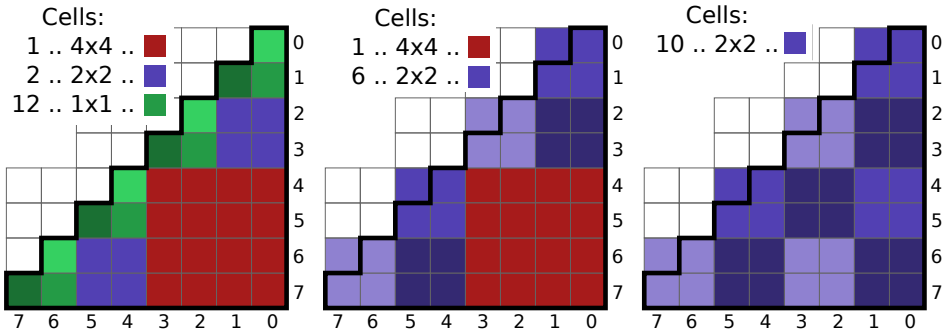


Figure 5.3: Categorization of baselines

In most of the experiments we use a simple single cell size tiling which does not cover all the baselines. This is acceptable when using small cells which cover most of the triangle and leave only small number of missing baselines. Since the number of missing baselines is small, we expect the difference in performance between calculating all the baselines and calculating only the regular baselines to be negligible.

However, with large cell sizes we get a large number of missing baselines and the performance difference can not be neglected any more. In very few cases where the underlying architecture prefers very large cell sizes, we use *full tiling* which covers all the baselines. Normally, the presented experiments do not use full tiling. If they do, it is explicitly stated.

We identified three different possibilities to implement the full tiling. The first option (which we initially implemented) is to calculate the regular baselines first and then in the second step recursively create cells of different sizes as large as possible (Figure 5.4). To handle the second step one has two alternatives. The first one is to issue as many kernel calls as there are cell sizes. This is not efficient since each kernel call would have only very few cells to calculate, which would lead to a poor device utilization. The second alternative (which was also implemented) is to create a flexible kernel that can calculate in a single call cells of different sizes. Nevertheless, the flexibility of the kernel to calculate cells of different sizes caused that the compiler was not able to take advantage of the cells and store them in the registers, which resulted in a low performance.



(a) Option 1 (b) Option 2 (c) Option 3

Figure 5.4: Different options for calculating missing baselines

Our second attempt to implement the full tiling creates an initial non full tiling first and then in second step creates complementary tiling for the missing baselines which can exceed the calculated triangle (Figure 5.4). The complementary tiling uses cells of the same size which can be different from the cells for the regular baselines. Unlike in the previous option we do not have to create a super flexible kernel. We simply run our kernel for arbitrary cell size twice. As a result, the performance of full tiling is acceptable (see Figure 5.8).

Third option to implement full tiling is to calculate all baselines in one step. All the cells would be of one size and in order to cover all the baselines the tiling would exceed the calculated triangle. However, with large cells we would calculate many unwanted baselines (Figure 5.4) resulting in large overhead. Since we needed a solution for large cells, we did not implement this option.

5.3 Exploration of parameters' values

After constructing a tunable kernel we can proceed to perform the actual tuning. We tune the kernel in two steps. First, we explore the parameters separately to get a feel of how they individually influence the performance. We also hope to use this separate tuning to find out border values beyond which the performance drops dramatically. Afterwards, we perform combined parameter tuning on parameters that correlate.

To reliably measure the performance of different parameter values, we chose GFLOP/s as our unit of measure. Simply measuring time could be highly imprecise as cell size and shape parameters influence the number of operations exe-

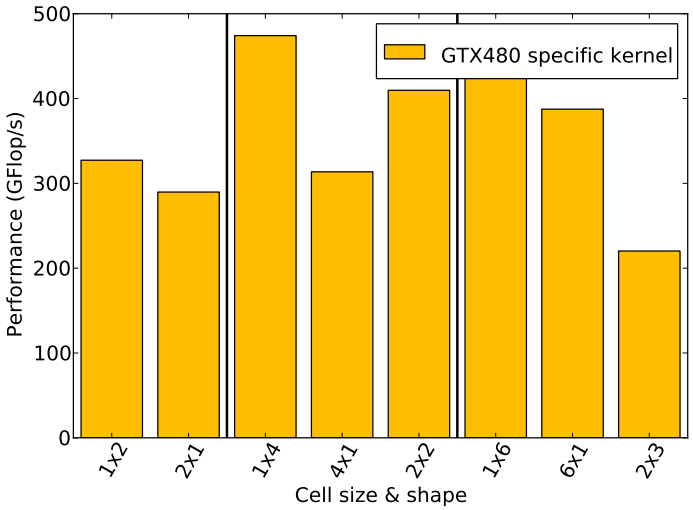


Figure 5.5: Cell shape tuning on GTX 480 (higher is better)

cuted. In the case of the full tiling we calculate also unwanted baselines Therefore it is questionable which operations should be included when calculating the performance. We can either include the operations of the unwanted baselines or we can leave them out including only the operations of the baselines within the triangle. As the purpose of the correlator is to calculate only the baselines within the triangle, we are more interested in the performance of those baselines than the performance of all calculated baselines. As a consequence, when mentioning the performance of full tiling we only include operations required to calculate the regular baselines.

5.3.1 Cell shape

The cell size and shape are parameters with great performance impact. The larger the cell is the more stations can be stored in registers and the less accesses to global memory occur. However, with too large cells there is register spilling which often leads to a significant performance drop.

The impact of the cell shape is best seen in Figure 5.5 which shows that GTX 480 prefers a cell to be a vertical line. We attribute this behavior to a convenient memory access pattern in which each thread has to load only one new station. On the other hand, with the horizontal or square cells each thread must read multiple new stations.

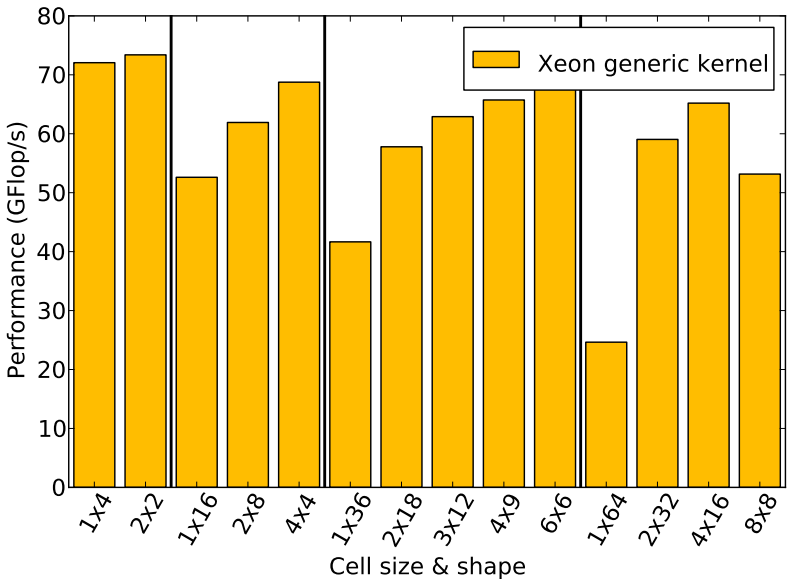


Figure 5.6: Cell shape tuning on Xeon (higher is better)

Contradictory to GTX 480, Intel Xeon performs better with square cells (see Figure 5.6). There are two factors which we believe make square cells most suitable for Xeon. First of all, CPUs can take advantage of their larger and better caches which makes them less sensitive to memory access patterns [26]. Secondly, the use of registers is better with square cells than with any other shape. For example, the cell $1x4$ loads 5 stations into registers while the cell $2x2$ needs to load only 4 stations.

5.3.2 Cell size

Using the best shape inferred in the previous section, we examine the impact of the cell size on the correlator performance.

As Figure 5.7 suggests, the GTX 480 is quite sensitive to the used cell size. We can see that for larger tile sizes (from 1x7) the performance drops dramatically due to register spilling. One can also notice that the optimal cell size for 32 and 64 stations is different. 32 stations have only a small number of cells; for instance, for the cell size of 1x2 we have only 256 cells. Given that one work item calculates one cell, and that for the experiment in Figure 5.7, we used work group size of 256 we need at least 256 cells to fully utilize the device. To get so many cells we

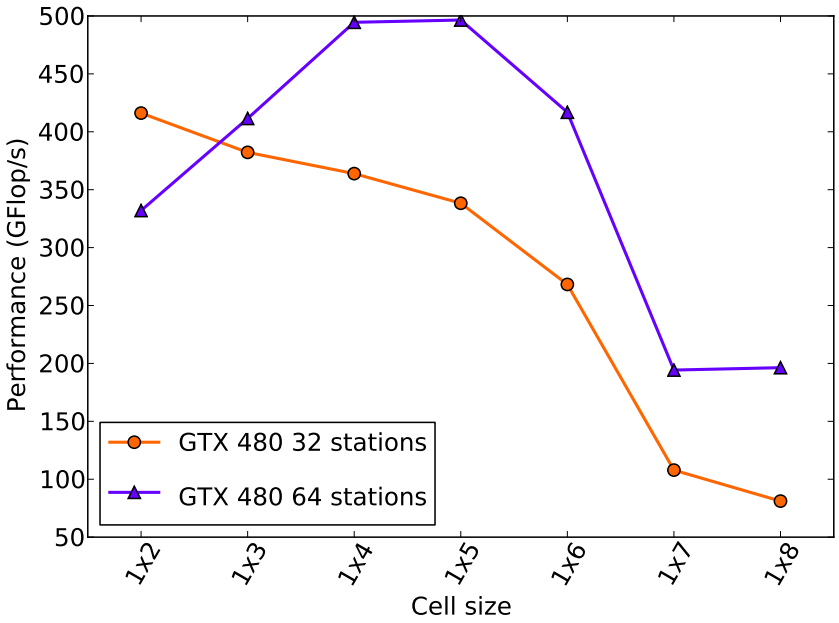


Figure 5.7: Cell size tuning on GTX 480 with work group size 256 (higher is better)

need the single cell to be of size 1x2 or smaller. Any larger cell would result in the number of cells smaller than 256 (e.g. 1x3 results in 165 cells); thus, we get a large number of stalled threads which are not doing any work.

The AMD Magny-cours, behaves quite contradictory to GTX 480. Judging from Figure 5.8, the Magny-cours is much less sensitive to the cell size, and it even performs better when using larger cells. However, with larger cells we also get fewer cells. More specifically, when correlating 64 stations, we get only one cell of size 32x32 resulting in 1056 missing baselines from a total of 2080. Hence, in order to safely proclaim that Magny-cours performs best under larger cells it is necessary to perform the experiment with full tiling scheme (Figure 5.8) which in the end proves that the Magny-cours performs better with larger cells. We suppose that this is happening due to two large L3 caches with a total size of 12 MB.

An experiment run on the Intel Xeon (Figure 5.9), confirms the theory that CPUs are less sensitive to cell size. The Xeon performs best with cells of size 2x2. The larger cells give comparable performance which is (with a few exceptions) almost constant. As it is outside of the scope of this thesis, we did not investigate the nature of the exceptions.

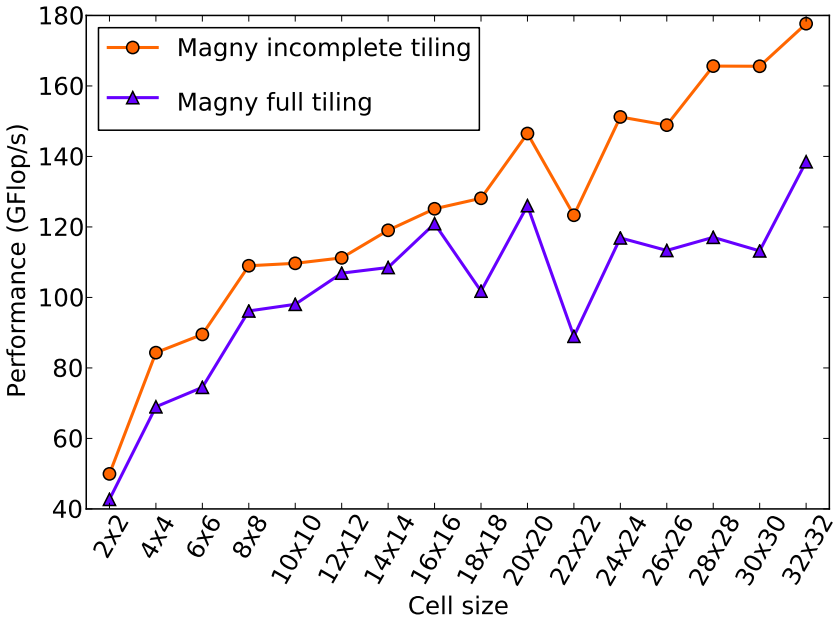


Figure 5.8: Cell size tuning on Magny-cours with 64 stations and work group size 50 (higher is better)

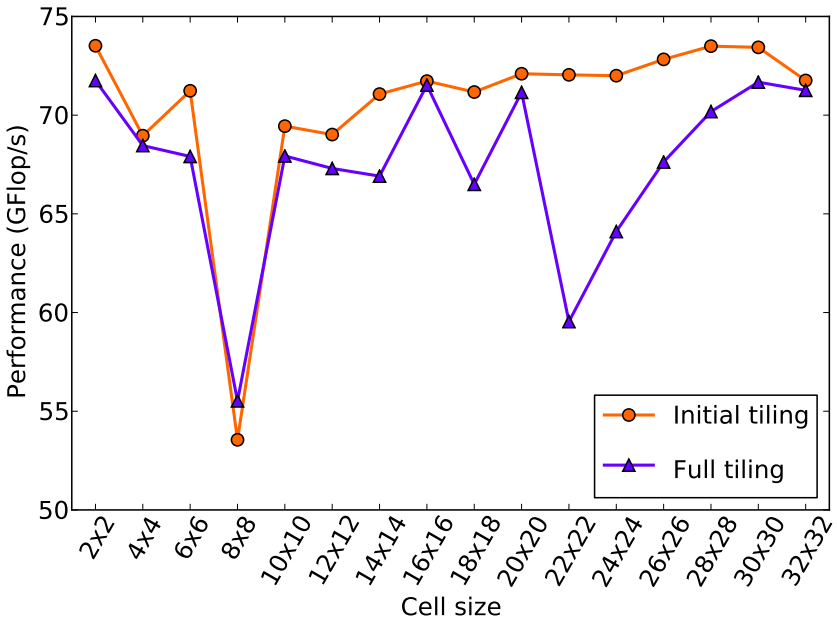


Figure 5.9: Cell size tuning on Intel Xeon with 64 stations and work group size 50 (higher is better)

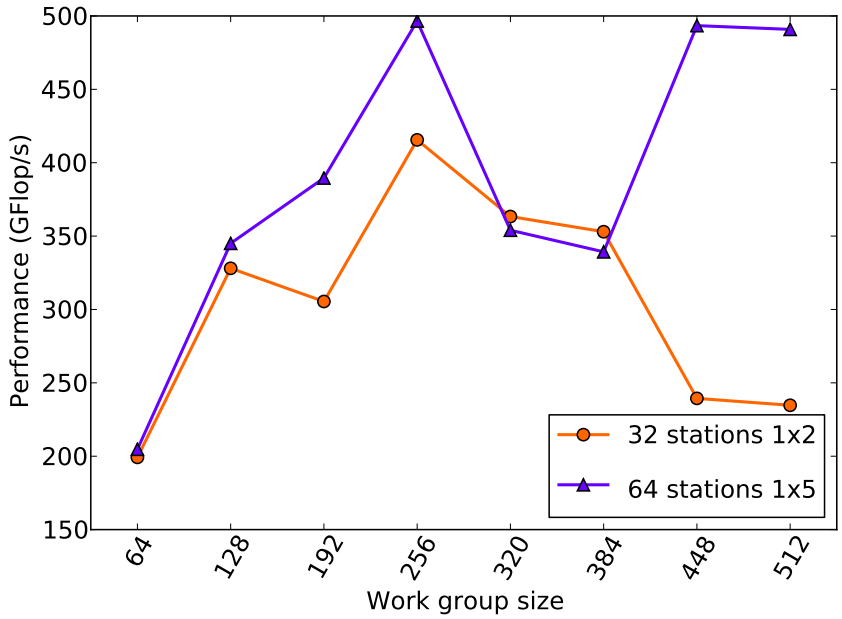


Figure 5.10: Work group size tuning on GTX 480 (higher is better)

5.3.3 Work group size

In this section we take the best cell size and shape found in the previous two sections, and tune the work group size.

Again, the GTX 480 is quite sensitive to the chosen work group size. As Figure 5.10 illustrates, with given cell size and shape, it achieves its peak at 265 work items per work group. The Intel Xeon is again much less sensitive than the GTX 480 and achieves almost constant performance (plus or minus 2 GFLOP/s) no matter whether the work group size is 16 or 512. AMD Magny-cours behaves similarly. We attribute this behavior to OpenCL thread mapping on CPUs, where it does not really matter whether we have 256 work items per work group and 16 work groups, or 16 work items per work group and 256 work groups.

5.3.4 Shared memory

When running a kernel on a GPU, enabling the shared memory is not really an auto-tuning parameter. It is an optimization which in some cases brings performance improvement. Furthermore, it forces the programmer to think about data

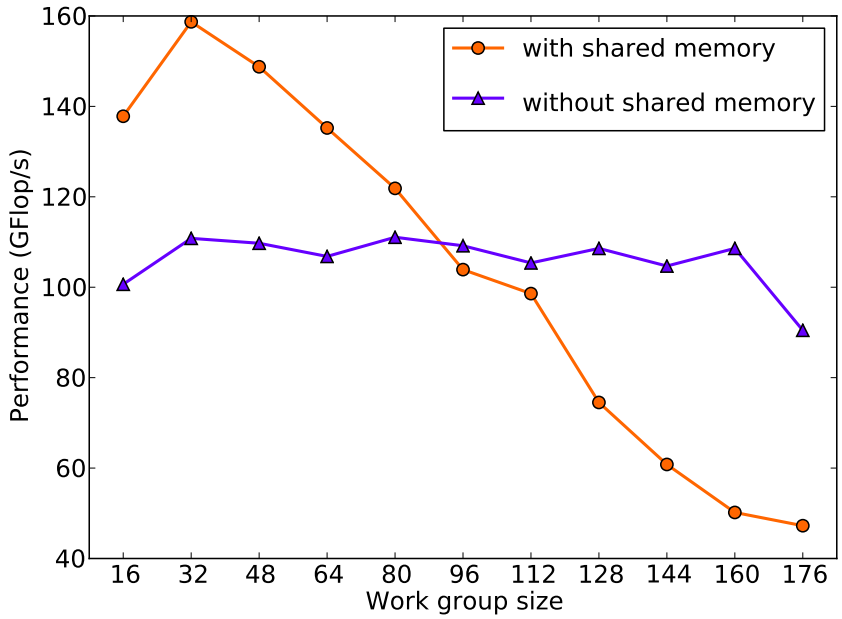


Figure 5.11: Shared memory tuning on AMD Magny-cours with cell size 8x8 (higher is better)

locality through which one can achieve better caching behavior. However, on CPUs it is less clear whether to enable the shared memory or not. Usually, the compiler stores the shared memory in global memory. Therefore, enabling the shared memory results in redundant memory copying. However, sometimes the compiler tries to store the shared memory in registers rather than in global memory; hence, improving the performance. For example, on the Magny-cours, the use of shared memory in combination with a smaller work group size yields better results than omitting the shared memory (see Figure 5.11). Unfortunately, when using the shared memory, we can only run the kernel on cells of sizes up to 12x12 (when correlating 64 stations). When calculating larger cells, the kernel crashes, most probably due to an inability to allocate all the necessary resources. On the Intel Xeon the use of shared memory always decreases the performance (Figure 5.12), probably because the compiler allocates the shared memory in global memory space.

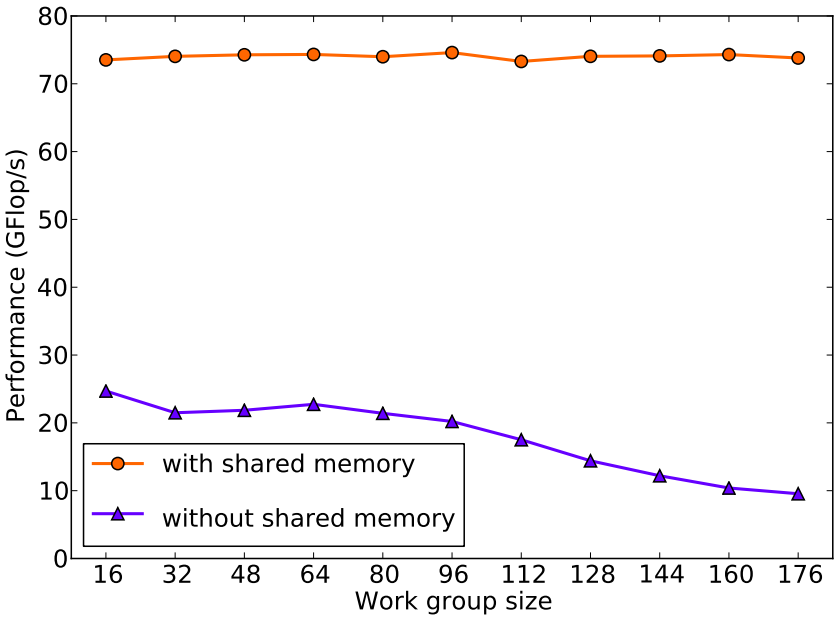


Figure 5.12: Shared memory tuning on Intel Xeon with cell size 2x2 (higher is better)

5.3.5 Combining several parameters

For the correlator kernel, it is essential to combine the auto-tuning of the cell size and the work group size. The cell size influences how many cells we have and the work group is responsible for calculating those cells. Ideally, we would like to have the work group divisible by the number of cells, so that each work item from a work group calculates one or more cells. That means that we would like to avoid the situation where half of the work group calculates two cells and the other half only one cell which leads to divergent and stalled threads. Since the optimal work group size is also hardware and kernel dependent, it is not enough to simply set the work group size to the number of cells. Thus, the combined auto-tuning is inevitable.

In order to shorten the time of the auto-tuning we limit the search space using the knowledge gained through separate parameter tuning in previous sections. For example, we limit the size of the cell on GPU to a maximum of 1x7, and we do not explore larger cells.

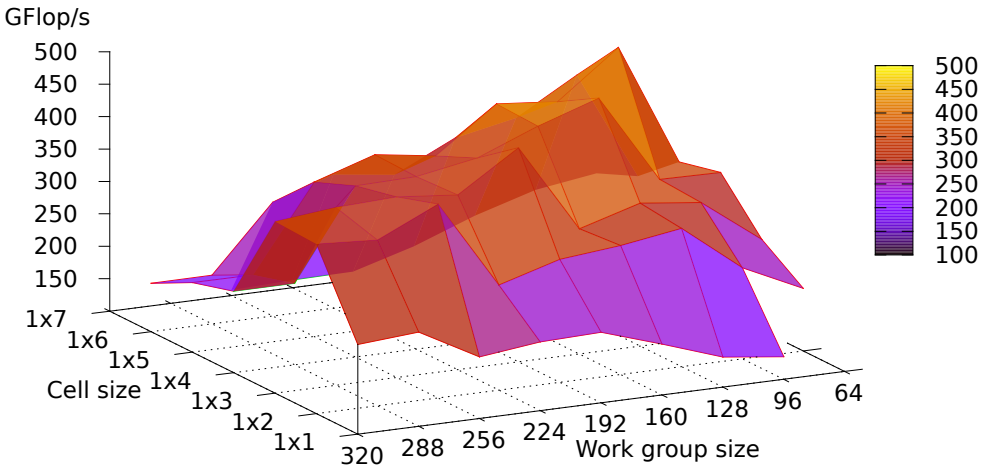


Figure 5.13: Cell size and work group size tuning on GTX 480 for 32 stations (higher is better)

So far the best achieved performance on the GTX 480 for 32 stations is 416 GFLOP/s and uses a cell of size 1x2 and 256 as the work group size. For 64 stations the best performance, 496 GFLOP/s, is with a cell of size 1x5 and 256 as work group size (see Figure 5.10).

However, as depicted in Figures 5.13 and 5.14, tuning the the cell size together with the work group size on GTX 480 gives for 32 stations 485 GFLOP/s with 96 work items per work group and cell of size 1x5. For 64 stations it is 545 GFLOP/s with 320 work items per work group and a cell of size 1x6.

Tuning the cell size together with the work group size on the Intel Xeon gives no new results and confirms the result from Section 5.3.3 that the Xeon is indifferent to work group size. Consequently, the Intel Xeon achieves the best performance of around 70 GFLOP/s for a cell size of 2x2 (Figure 5.9).

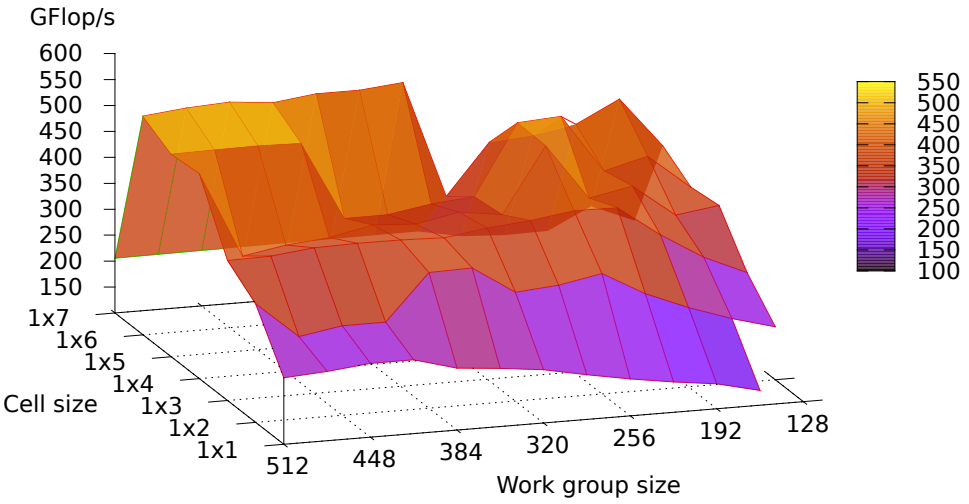


Figure 5.14: Cell size and work group size tuning on GTX 480 for 64 stations (higher is better)

On the AMD Magny-cours it does make sense to tune the cell size together with the work group size as long as the shared memory is turned on (without the shared memory the work group size has only subtle influence on performance). As Figure 5.15 illustrates, we achieve the best performance, little above 190 GFLOP/s, with the cell of size 12x12 and 10 work items per work group. This result is slightly better than the previous result of tuning the cell size alone with fixed work group size and no shared memory (Figure 5.11).

Multiple experiments in this section reveal that when using the shared memory, the optimal work group size is equal or a little higher than the number of cells. For CPUs the best value of the work group size is the number of cells. In case of GPUs the optimal work group size is equal or little higher than the number of cells and divisible by 32 (the number 32 was chosen according to the suggestions in [24]).

All in all, this section proves that the tuning of combinations of parameters can give better results than tuning each parameter separately.

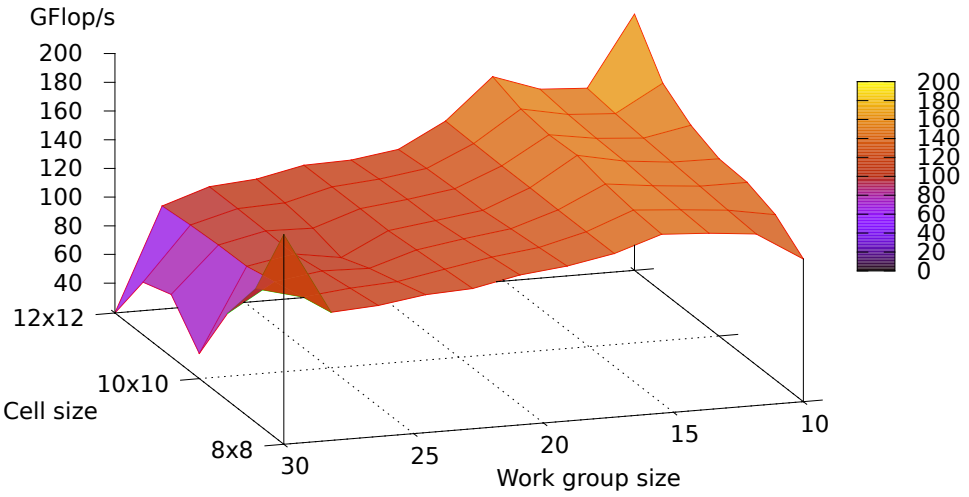


Figure 5.15: Cell size and work group size tuning on AMD Magny-cours (higher is better)

5.4 Conclusions

By employing auto-tuning, we successfully found optimal parameters' values for three different architectures, and we can easily apply our approach to other architectures as well.

In the best case scenario we were able to achieve the same performance as the hand-tuned kernel. In suboptimal scenarios our auto-tuned kernel even exceeds the performance of the hand-tuned one. This is shown in Figure 3.2 from chapter on programming model (Section 3.3) which compares our auto-tuned OpenCL implementation to a hand-tuned CUDA implementation.

Additionally, judging from our observations and experiments, we can conclude the following about auto-tuning:

- **Auto-tuning matters**
  Since GPUs are very sensitive to memory access, occupancy and divergent threads, a subtle change in parameter values can result in a substantially large change in performance. Consequently, the ability to explore different parameter values is essential. On the other hand, CPUs are much less

sensitive as a small change in the parameter value often results in a negligible performance change. Nevertheless, the optimal values for a CPU are still very different from those for a GPU; thus, auto-tuning is still desirable. Even more so if we take into account the fact that the performance of different OpenCL optimizations such as use of shared memory is much less predictable than for GPUs.

- **Auto-tuning requires less prior knowledge**
  When compared to hand-tuning, auto-tuning requires less knowledge about the used hardware. For example, we do not have to know the number of registers or the number of compute cores to choose the correct cell size or work group size as auto-tuning will do that. On the other hand, prior architecture knowledge helps to prune the search space.

- **Studying auto-tuning results brings more knowledge**
  By studying auto-tuning results, we can gain a deeper understanding of our kernel and the hardware it is ran on (as presented in the previous sections). For example, if we did not have a tunable kernel which would be able to explore all cell shapes and sizes, we would never find out the preferred cell shape of the different architectures.

- **Auto-tuning does not come for free**
  Designing a general and flexible kernel suitable for auto-tuning takes time and effort. Furthermore, additional time needs to be invested when performing the tuning. However, we believe that the time invested in designing a flexible kernel is considerably shorter than hand-tuning the kernel for different platforms.

- **Auto-tuning must be performed wisely**
  Exploring all the possible values and combinations is often unfeasible; hence, it is convenient to first tune each parameter alone to find some boundaries (i.e. those which, if crossed, always cause a significant drop in performance). Afterwards, one can use these limits in the combined parameter tuning to limit the search space. It is also helpful if one can identify parameters which do not correlate with any other parameters and are safe to be tuned separately.

Chapter 6

Data layout conversion

In this chapter we first introduce the notion of data layout (Section 6.1). Since different kernels favor different layouts, it is desirable to auto-tune the input and output data layouts of each kernel. However, to enable the data layout tuning, we need to have a possibility to convert from one layout to another. As a result, two conversion schemes are analyzed in Section 6.2. Their implementation and performance behavior is presented in Section 6.3. Finally, Section 6.4 compares the two schemes on a real life example and chooses the better one for future experiments.

6.1 Data layout introduction

In our radio astronomy pipeline, a single input sample is a complex number representing amplitude and phase of a signal. Each sample is further characterized by:

- The telescope station which observed the sample.
- The frequency channel at which the sample was observed.
- The polarization of the signal.
- The time stamp (we perform several observations in the time period of 1 second).

The above sample characteristics makes the pipeline data to be an n-dimensional array (with the following dimensions: station, channel, polarization and time).

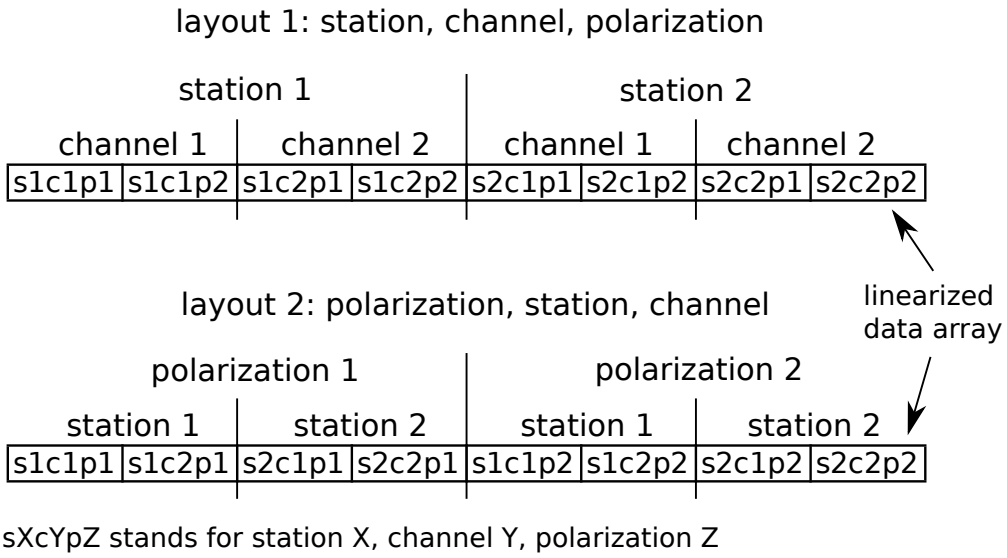


Figure 6.1: Two different layouts with dimensions: station, channel, polarization. (In our pipeline we would have in most kernels a fourth dimension: time).

The exact order of the dimensions specifies the data layout. In the following text we often abbreviate the layout by only stating the first letters of the dimensions. For example, CSTP stands for channel, station, time and polarization. The difference between two layouts is illustrated in Figure 6.1. Although, logically, the data is n-dimensional, physically, it is stored as a single one dimensional array. Since each kernel reads and writes data, we distinguish between read and write or input and output data layouts for each kernel.

The used data layout determines the access pattern of the kernel, and thus it has crucial influence on the performance of the kernel. Since different kernels have different optimal input and output data layouts, a mechanism for converting from one layout to another is necessary. The conversion from one layout to another can be viewed as several subsequent dimension transpositions; hence, we call this conversion a *transposition* or *transpose operation*.

6.2 Transposition schemes analysis

Suppose two kernels, kernel K1 and kernel K2 which are connected to each such that K1 is followed by K2. Furthermore, assume that the optimal output data layout of K1 is different from the optimal input data layout of K2 (i.e. a transposition

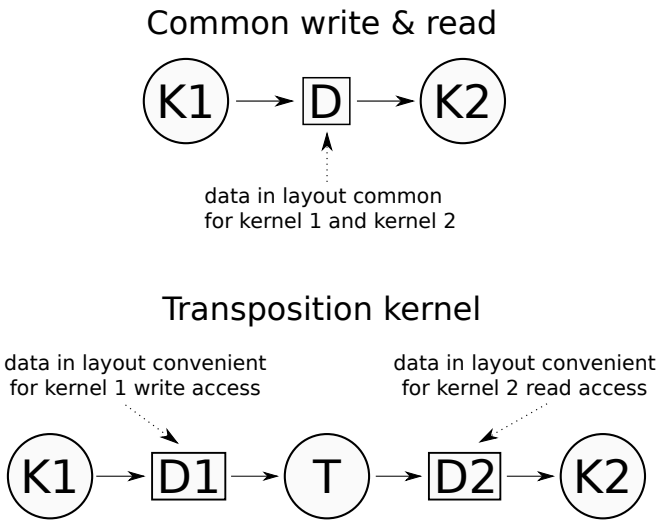


Figure 6.2: Transposition schemes

is required). We have two basic possibilities to implement the transposition:

- Common write & read: there is a predefined layout in which K1 writes the data and K2 reads it.
- Transposition kernel: a special transposition kernel is placed between K1 and K2 enabling K1 to write and K2 to read the data in their respective optimal data layouts.

The two options are depicted in Figure 6.2 and compared in Table 6.1. The greatest advantage of the transpose kernel is its usability with third party kernels whose output layout we can not change. For example, when using a third party FFT library, we need the transposition kernel to adjust the data layout for kernels following the FFT kernel. The main disadvantage of the transposition kernel is due to the 4 required accesses to the global memory instead of the 2 required accesses of the common write & read. To illustrate the situation, we mark the transposition kernel as T and count the reads and writes: K1 writes data, T reads and subsequently writes data and K2 reads data. On the other hand, the common write & read has only 1 write (K1) and 1 read (K2).

To be able to use the flexibility of the transposition kernel, we want to find out under which circumstances the transposition kernel outperforms the common write & read. The theoretical formulas showing the execution time decomposition of

transposition scheme	optimal read & write	usable with 3rd party kernels	memory access	extra memory
transposition kernel	yes	yes	4	yes*
common write & read	not guaranteed	not guaranteed	2	no

Table 6.1: Comparison of the transposition schemes. The left side of the table shows advantages of transposition kernel whereas the right side shows the advantages of the common write & read.
* In case of in-place transposition the extra space is not needed. However, out-of-place transposition is more likely to achieve better performance; hence, we count in the extra memory.

the two options follow:

$$T^c = \underbrace{T^c_{k_1}}_{T^c_{r_1}+T^c_{e_1}+T^c_{w_1}} + \underbrace{T^c_{k_2}}_{T^c_{r_2}+T^c_{e_2}+T^c_{w_2}}$$

$$T^t = \underbrace{T^t_{k_1}}_{T^t_{r_1}+T^t_{e_1}+T^t_{w_1}} + T_t + \underbrace{T^t_{k_2}}_{T^t_{r_2}+T^t_{e_2}+T^t_{w_2}}$$

where

c	common write & read scheme
superscript t	transposition kernel scheme
subscript t	the actual transpose kernel
k1 or 1	kernel K1
k2 or 2	kernel K2
r	read
e	execute
w	write

To compare the times of the two transposition schemes we can leave out the times which are the same for both schemes. Consequently, the formulas can be reduced to:

$$T^{c\prime} = T^c_{w_1} + T^c_{r_2}$$
$$T^{t\prime} = T^t_{w_1} + T_t + T^t_{r_2}$$

Furthermore, the transposition kernel scheme allows the kernel K1 to write the data in the optimal layout (T_{ow_1}) and the kernel K2 to read the data in the optimal layout (T_{or_2}). Thus, we can change the formula for the transposition scheme to:

$$T^{t\prime} = T_{ow_1} + T_t + T_{or_2}$$

It follows that the transposition kernel scheme outperforms the common write & read if the following formula is satisfied:

$$T_t < \underbrace{(T^c_{w_1} - T_{ow_1})}_{\text{K1 write overhead}} + \underbrace{(T^c_{r_2} - T_{or_2})}_{\text{K2 read overhead}} \tag{6.1}$$

Since it is hard to measure the read and write times alone, we compare the total kernel times:

$$T_t < \underbrace{(T^c_{k_1} - T_{k_1ow})}_{\text{K1 write overhead}} + \underbrace{(T^c_{k_2} - T_{k_2or})}_{\text{K2 read overhead}} \tag{6.2}$$

where T_{k_1ow} is the total time of K1 with optimal write and T_{k_2or} is the total time of K2 with optimal read. Expecting that the K1 execution and read times are the same for both transposition schemes ($T^c_{r_1} + T^c_{e_1} = T^t_{r_1} + T^t_{e_1}$) and analogically the K2 execution and write times are the same ($T^c_{e_2} + T^c_{w_2} = T^t_{e_2} + T^t_{w_2}$), the conditions 6.1 and 6.2 are the same.

In conclusion, translating Formula 6.2 to words, there are two requirements for the transposition kernel to outperform the common write & read:

- The optimal common K1 write and K2 read layout must result in positive K1 write or K2 read overhead.
- The transposition kernel must outperform the overhead.

Further on, in Section 6.4 we show that it is possible to satisfy Formula 6.2.

6.3 Transposition schemes implementation

In order to implement the common write & read scheme, we need a mechanism which would allow each kernel to read and write from the global memory in any layout. To achieve this we use the OpenCL feature of run time compilation. First, we specify for each kernel its read and write layout in the form of program parameters. Then, during runtime, based on the given read and write layout of the kernel, we generate the specific OpenCL code for obtaining input and output data indices. An OpenCL fragment for reading layout CTS and writing layout STC is presented in Algorithm 1.

To be able to compare the common write & read to the transposition kernel, we have implemented the transpose kernel. Our implementation assigns one thread

Algorithm 1 Generated OpenCL read and write index calculations

```
1: // Input index calculation for layout channel, time, station
2: #define INPUT_IDX(channel, time, station)
3:     (channel * TIMES + time) * STATIONS + station
4:
5: // Output index calculation for layout station, time, channel
6: #define OUTPUT_IDX(channel, time, station)
7:     (station * TIMES + time) * CHANNELS + channel
```

to convert one sample; hence, we create as many threads as there are samples. The basic idea which illustrates how the transpose kernel converts one index to another is illustrated in Algorithm 2. Lines 3 - 9 are just a generic form of index calculations present in Algorithm 1. The function *getIndices()* is the reverse calculation which extracts from the global index and dimension sizes the dimension indices. For a better explanation of the transpose kernel, let us suppose that we want to convert between layouts CTS and STC (present in Algorithm 1). Then the variables used by the transpose kernel (dimensionIndices, dimensionSizes and permutation) would contain the following values:

dimensionIndices	[channel, time, station] - calculated from global thread ID
dimensionSizes	[CHANNELS, TIMES, STATIONS]
permutation	[3, 2, 1] -

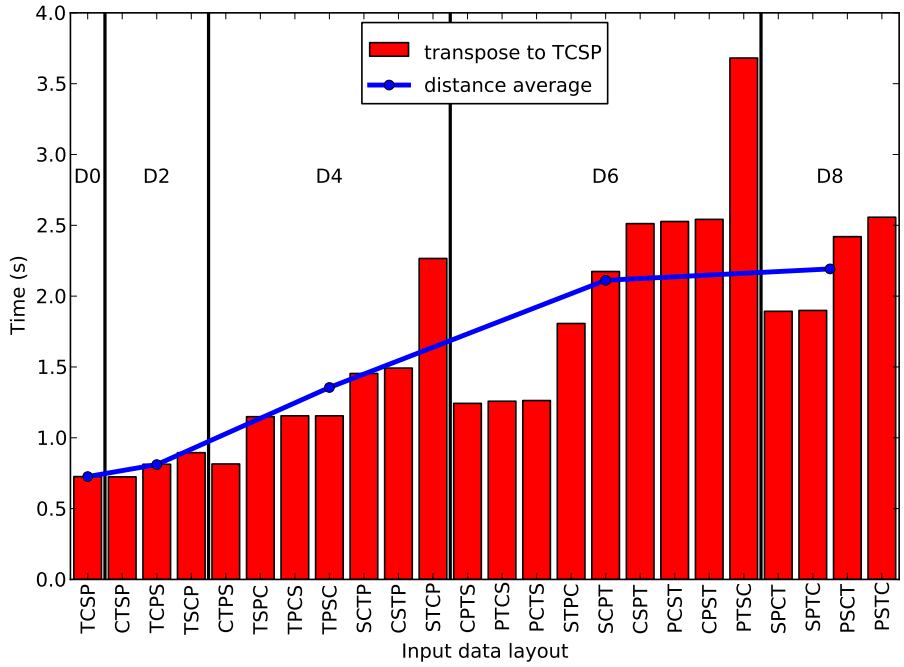
Algorithm 2 Simplified version of the transpose kernel

```
1: int inIdx = getGlobalId(0)
2: int[] dimensionIndices = getIndices(inIdx, dimensionSizes)
3: int outIdx = dimensionIndices[permutation[0]]
4: for i from 1 to DIMENSIONS do
5:     outIdx = outIdx *
6:         dimensionSizes[permutation[i]] +
7:         dimensionIndices[permutation[i]];
8: end for
9: output[outIdx] = input[indIdx]
```

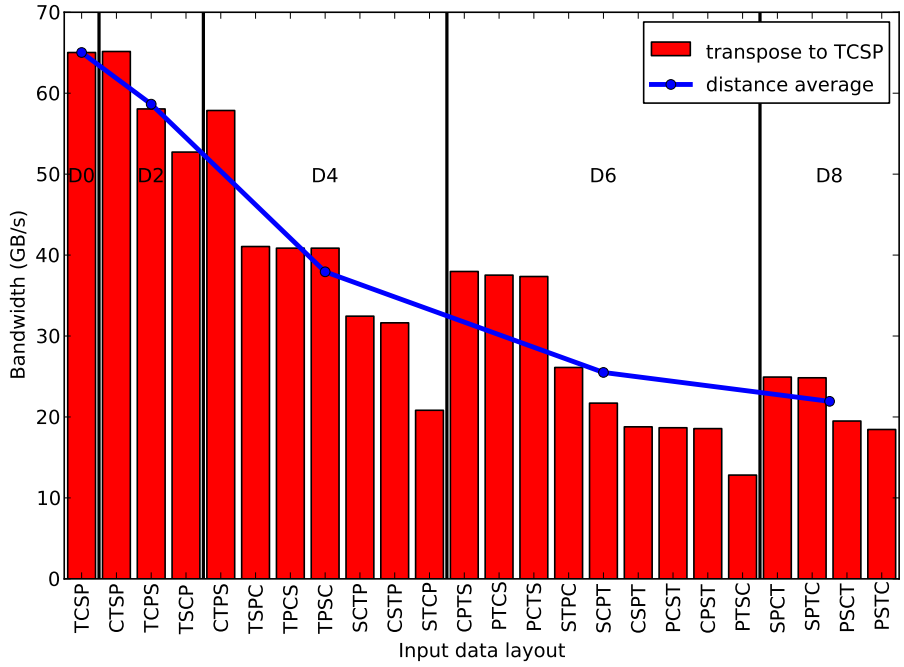
The read access pattern of the transpose kernel is coalesced: each consecutive thread reads next sample from the input array. However, the write access pattern depends on the layout to which we are converting. We assume that the greater the difference between the two layouts is, the worse the write access pattern of

the transpose kernel is. To measure the difference between two layouts, we introduce a property called distance. Having two different layouts containing the same dimensions, the distance is calculated as sum of position difference of each dimension. For example, the distance between CTS and STC is 4 since both C and S are 2 positions away from their position in other layout.

Figures 6.3 and 6.4 illustrate the relation between the transpose kernel performance and the layouts distance. While the experiment in Figure 6.3 was performed on a GPU, the experiment in Figure 6.4 was performed on a standard CPU. We can see that both CPU and GPU exhibit the same overall behaviour. The difference is just in CPU being less sensitive to the distance, most probably due to better caching. The blue average line suggests that layouts of lower distance are likely to perform better than layouts of larger distence but it does not prove it.

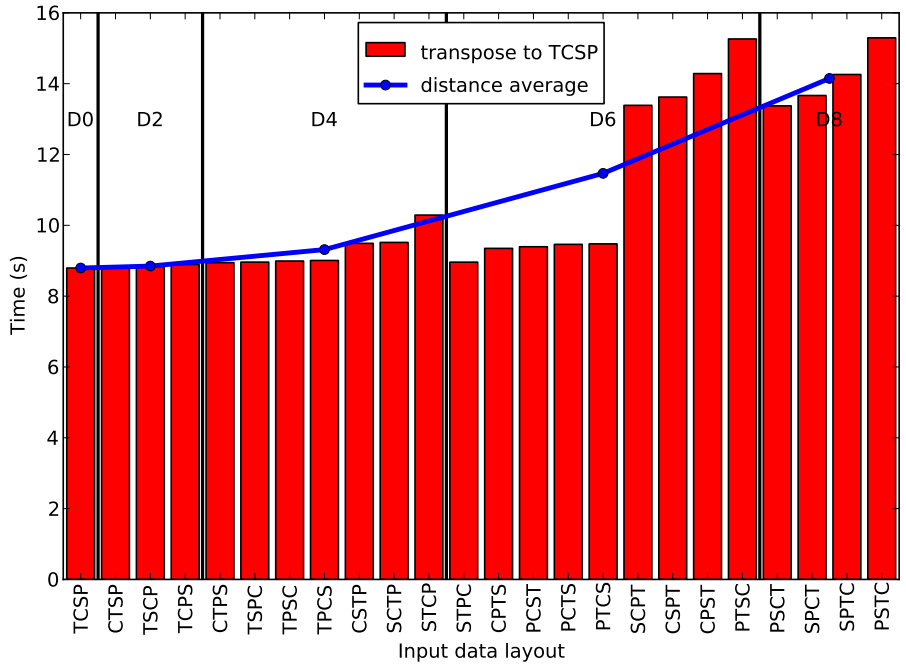


(a) Transpose time (lower is better)

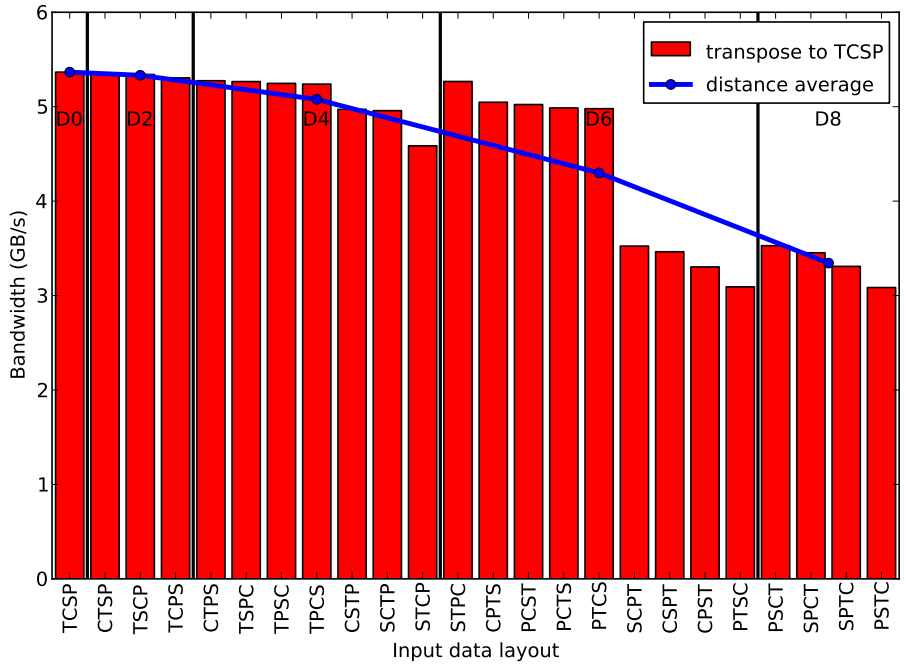


(b) Transpose bandwidth (higher is better)

Figure 6.3: Transposition from all possible layouts to TCSP measured on GTX 480. The gray vertical lines divide the distances 0, 2, 4, 6 and 8. The experiment uses 64 stations, 300 channels, 768 time samples and 2 polarizations.



(a) Transpose time (lower is better)



(b) Transpose bandwidth (higher is better)

Figure 6.4: Transposition from all possible layouts to TCSP measured on Intel Xeon. The gray vertical lines divide the distances 0, 2, 4, 6 and 8. The experiment uses 64 stations, 300 channels, 768 time samples and 2 polarizations.

low	$high$	$\lvert S_{low}\rvert$	$\lvert S_{all}\rvert$	P_{low}
2	4	20	21	0.95
4	6	52	63	0.83
6	8	21	36	0.58

Table 6.2: Likelihood of a lower distance (low) outperforming a larger distance ($high$) according to Formula 6.3 and Figure 6.3 (GTX 480).

We can measure the likelihood of a lower distance performing better than a larger distance exactly with the following formula:

$$P_{low} = \frac{|S_{low}|}{|S_{all}|} \tag{6.3}$$
$$S_{all} = \{\forall(T_{low}, T_{high})\}$$
$$S_{low} = \left\{\forall(T_{low}, T_{high})_{T_{low}<T_{high}}\right\}$$

where T_{low} stands for performance (measured in time) of any lower distance layout and T_{high} represents performance of any larger distance layout (values of low and $high$ are fixed). The set (T_{low}, T_{high}) represents all possible performance pairs of lower and larger distance. If we choose arbitrarily two layouts of different but fixed distance, then Formula 6.3 expresses the probability of the lower distance layout outperforming the larger distance layout. When we take the values from Figures 6.3 and 6.4 and substitute them to Formula 6.3, we always get a result greater than 0.5 (for any two combinations of low and $high$). Specific values for interesting combinations of low and $high$ are presented in Table 6.2. Consequently, the statement that a lower distance layout is likely to perform better than a larger distance layout is justified. This finding will be used in next section (6.4) to improve the overall performance of our specific pipeline scenario.

The search for an optimal transpose kernel, possibly also using shared memory and doing in place transposition[19], is a complex topic on its own, and as such should be investigated further in a future work. Nevertheless, any kind of reasonably fast transpose kernel (such as that of ours) is an indispensable tool as it allows us to perform flexible data layout tuning.

6.4 Transposition schemes comparison

We will compare the performance of the two transposition schemes in a pipeline consisting of the FIR filter followed by the correlator [1]. To get the best performance of common write & read we need to find the minimum of (FIR write + correlator read) over all possible common data layouts. To get the best performance of the transpose kernel we need to find the minimum of (FIR write + transpose + correlator read) over all possible FIR write layouts and all possible correlator read layouts. To limit the search space we restrict ourselves to the best correlator read layout. The results of both searches are illustrated in Figure 6.5. The best performance of common write & read is 3.73s while the transpose kernel performs better and reaches 3.57s.

Having the actual times gathered in Figure 6.5, we can now relate to Formula 6.2. Since the first search displayed in Figure 6.5 contains all possible correlator read layouts, it also contains the optimal layout. Furthermore, since the second search uses the best correlator read layout from the first search, the correlator read overhead is 0 ($T^c_{k_2} - T_{k_2 or} = 0$). As a result Formula 6.2 reduces to:

$$T_t < \underbrace{(T^c_{k_1} - T_{k_1 ow})}_{\text{FIR write overhead}} \tag{6.4}$$

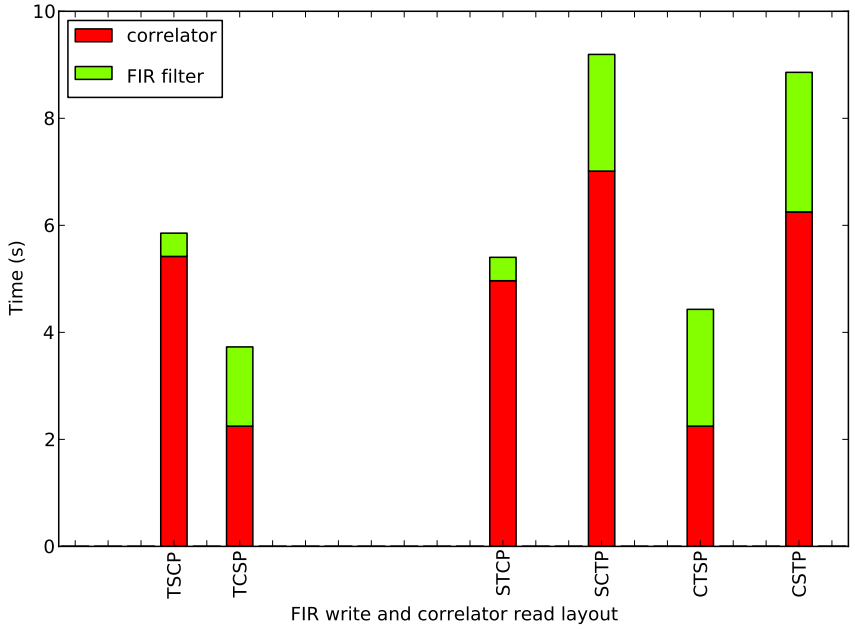
The time $T^c_{k_1}$ can be inferred from the first search (Figure 6.5a). It is the FIR time for the layout which gives the lowest FIR time + correlator time (1.481s).

The time $T_{k_1 ow}$ can be inferred from the second search (Figure 6.5b) as the best FIR time. There are several very good FIR times which are around 0.42s. If we choose exactly the best time (0.412s for layout PTSC or PSTC), we get the following time restriction on the transpose kernel:

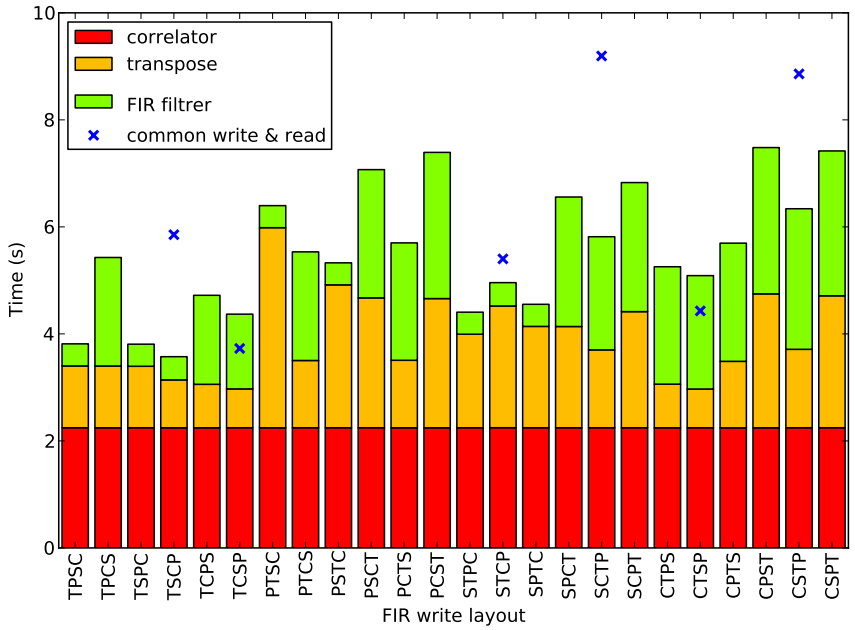
$$T_t < \underbrace{(1.481 - 0.412)}_{1.069}$$

However, since the FIR write layouts PTSC and PSTC are both distant from correlator read layout TCSP (distance 6 and 8), our transpose kernel performs poorly and takes more than 2s (see Figure 6.3). In order to satisfy Formula 6.4, we need a suboptimal layout which still gives a good FIR write time and which is closer

[1] In the real life pipeline there is always an FFT right after the FIR filter. However, since we use a third party FFT library, we can not transpose the data between the FFT and the correlator with the common write & read transposition scheme. Consequently, we leave out the FFT step in this particular experiment.



(a) Common write & read scheme.



(b) Transposition kernel scheme. The correlator read layout is fixed at time, channel, station and polarization. The blue crosses display a projection of Figure a.

Figure 6.5: Comparison of transposition schemes on GTX 480 (lower is better). The experiment was performed with 64 stations, 300 channels and 768 time samples. The FIR filter kernel uses 8 taps and 96 batches. The correlator kernel does not use shared memory.

to TCSP. For instance, the layout TSCP causes the FIR time to be 0.434s and it results in the transpose kernel time of 0.895s. Put together we get a condition:

$$\underbrace{T_t}_{0.895} < \underbrace{(1.481 - 0.434)}_{1.047}$$

which holds and thus proves that the theoretical Formula 6.2 can be satisfied in practice.

In conclusion, we showed that the transposition kernel scheme, in spite of doubling the number of global memory accesses, can outperform the common write & read. As a result, in the chapters that follow we will use the transposition kernel as it allows greater flexibility.

Chapter 7

Pipeline auto-tuning

In Chapter 5 we showed how to auto-tune a single kernel (the correlator). In this chapter we discuss the auto-tuning of the entire pipeline. By pipeline we mean a linear connection of multiple kernels. For example, the connection of the FIR filter, FFT and correlator kernels forms a pipeline.

To successfully tune a pipeline we first need to identify parameters which influence two or more kernels. In any kind of pipeline, one of the most important things influencing multiple kernels is the data that the kernels pass from one to another. Therefore, in this chapter we focus on the data layout tuning across our radio astronomy pipeline.

We start the chapter by stating where are we expecting performance improvements (Section 7.1). Afterward, Section 7.2 introduces the two specific pipelines that we tune and Section 7.3 lists the exact values of used parameters. Sections 7.4 and 7.5 are devoted to the actual tuning of our two pipelines during which we try to prune the search space as much as possible. We report shortly the pitfalls and results of a pipeline tuning on CPU in Section 7.6. Afterward, we study the efficiency of the search space pruning in Section 7.7. Finally, Section 7.8 concludes with the more general knowledge gained in this chapter.

7.1 Motivation

The main three kernels considered in this work (poly-phase filter, beam former and correlator) have all been extensively, one by one, manually tuned in previous

work [38, 39, 32]. Consequently, we consider them to be optimized for the data layouts they have when used separately. However, when used within a pipeline, conversions from one layout to another (transpositions) are sometimes inevitable. Hence, a layout optimal for a single kernel is not guaranteed to stay optimal within a pipeline where transposition is required.

By auto-tuning the data layout we expect to improve the overall performance of the pipeline by finding a layout that is suboptimal for its kernel but is optimal within the pipeline. Such layouts can exist between two kernels where transposition is required. In particular, we are looking for layouts that might slightly increase the kernel time but decrease the transposition time. A side effect of our search cane be that we find a layout better performing than the default one. By default layout we mean the original LOFAR data layout used in [38, 39, 32].

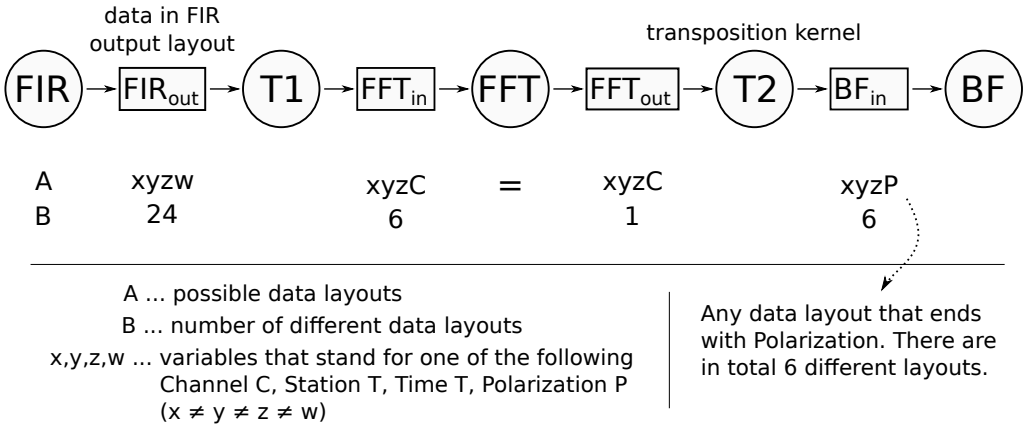
7.2 Specific pipelines analysis

We chose two specific pipelines for the auto-tuning. The first one (Pipeline 1) consists of the FIR filter, FFT filter and beam former (BF). It was chosen because it is one of the most frequently used pipelines in LOFAR. The second pipeline we chose (Pipeline 2) consists of the FIR, FFT, beam former (BF) and correlator (CR) kernels (we added the correlator kernel to Pipeline 1). Although Pipeline 2 represents a valid connection of astronomical kernels, it is hardly used and therefore is of a more theoretical nature. However, tuning Pipeline 2 is still very useful as it shows how a larger pipeline can potentially be decomposed into two smaller ones which can be tuned separately. Pipelines 1 and 2 are depicted in Figure 7.1.

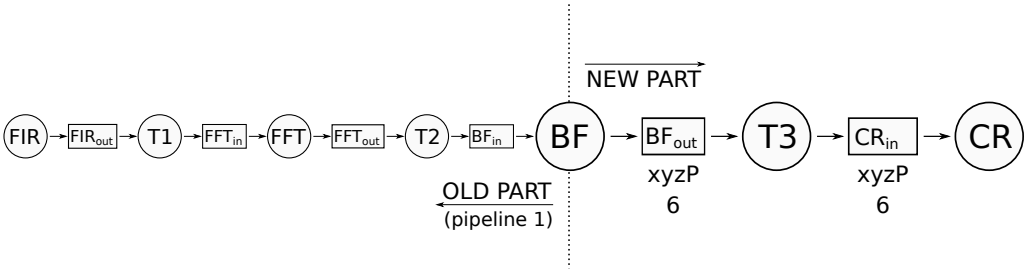
The samples which form the input of the kernels are characterized by the following numbers: channels, stations, time samples, polarizations and beams (generally called dimensions). The data layout is defined by the order in which these dimensions appear in the code. In the text that follows, we use 4 letters layout abbreviations such as TCSP which refers to the layout time samples, channels, stations and polarizations. For further information see *Data layout* chapter - Section 6.1.

By analyzing the data layouts in our two pipelines (Figure 7.1), we can make the following observations:

- While the transposition kernels T1 and T3 are optional, the transposition kernel T2 is necessary since the FFT write layout always ends with *channel* and the beam former read layout always ends with *polarization*.



(a) Pipeline 1: FIR, FFT and beam former.



(b) Pipeline 2: FIR, FFT, beam former and correlator.

Figure 7.1: The two explored pipelines together with all possible data layouts.

- Because we use a 3rd party FFT implementation, the read and write layouts are the same. Therefore, for a given FFT read layout there is always only one FFT write layout.
- There are all together $24 \times 6 \times 1 \times 6 = 864$ different data layout combinations in Pipeline 1 and $24 \times 6 \times 1 \times 6 \times 6 \times 6 = 31104$ combinations in Pipeline 2.

7.3 Experiments setup

In most of the experiments in this chapter we use the following pipeline configuration:

stations	64
channels	256
time samples	512
polarizations	2
iterations	100

While the number of stations, channels and polarizations reflect the typical LOFAR setup, the number of time samples is decreased from the usual 768 to 512 in order to have a power of two which is required by our FFT filter. Iterations refer to the number of times we repeat each kernel execution. We do this to artificially increase the kernel time to at least hundreds of milliseconds to get more valid time measurements. Furthermore, we use the following kernel configurations:

- **FIR filter**: 8 taps and 8 bit samples.
- **Beam former**: 100 beams and beam block size of 10 beams.
- **Correlator**: Cell size of 1x6.

All the experiments were performed on the GTX 480.

7.4 Tuning Pipeline 1

To get a better understanding of Pipeline 1 and consequently be able to prune the search space, we first tuned the FIR write layout, FFT layout and beam former read layout separately (Figure 7.2). Furthermore, to understand how much

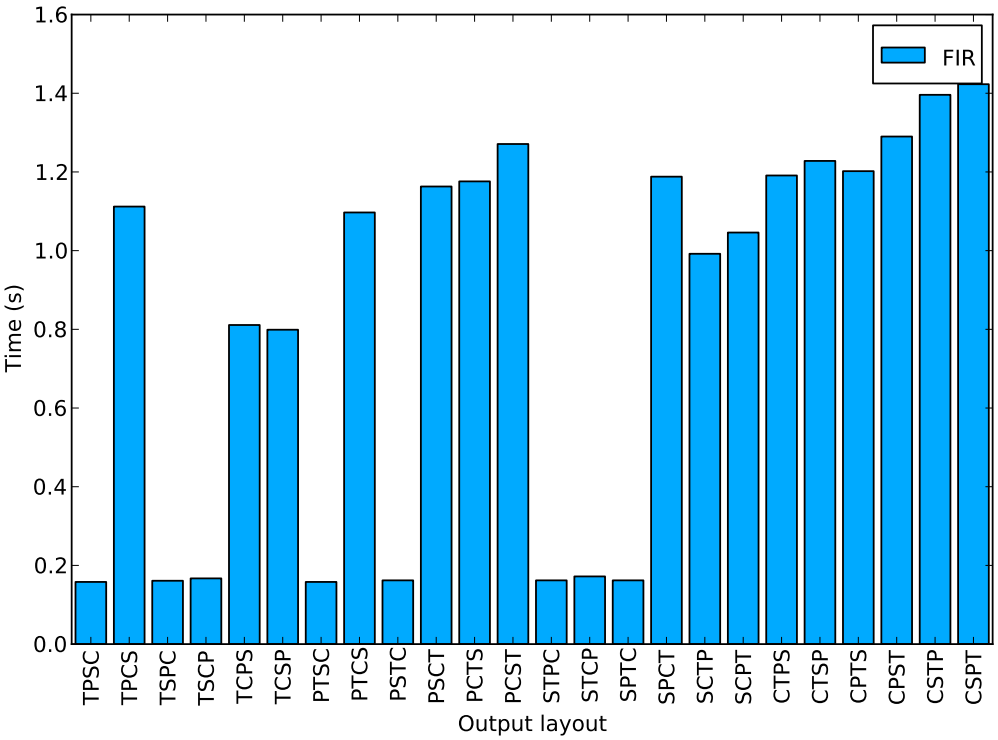
time we can save by doing a better transposition we also recorded the fastest and slowest transposition execution times which were 0.412 and 1.441 seconds, respectively.

From the kernel layout tuning we made the following observations:

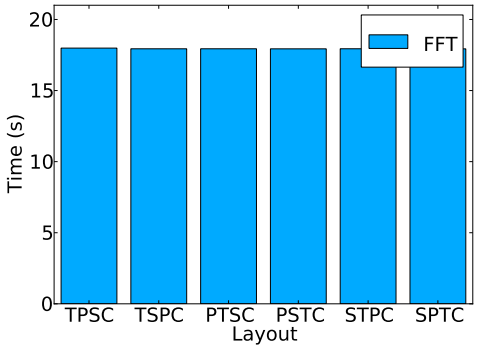
1. From 24 FIR write layouts there are 8 layouts which are considerably better (from 4 to 7 times) than the remaining layouts. Out of these 8 layouts the best 6 layouts end with *channel*.
2. All 6 FFT layouts have the same performance.
3. The difference between the best and worst transposition kernel layout is 1 second (i.e. by performing a better transposition, we can not save more than a second). For a better illustration, one second is 28% of the entire Pipeline 1 run (when excluding the slow FFT).
4. From 6 beam former read layouts there are 2 which are at least 3 times better (faster by more than 3.5 seconds) than the rest.
5. The 6 best FIR write layouts are very close in terms of performance. The same can be said about the 2 best beam former read layouts.

Based on observations 1 and 2, we can leave out the transposition kernel between FIR and FFT and reduce the combination of 24 FIR write layouts and 6 FFT layouts to only the 6 FIR write layouts which end with *channel*. Furthermore, using observations 3 and 4 we reduce the 6 beam former read layouts to only 2. As a result, the original 864 possibilities are now reduced to $6 \times 2 = 12$. To find the optimal layouts we would normally run the pipeline with all 12 layout combinations. However, observation 5 suggests that the performance of the 12 different layout combinations will only differ in the transposition kernel. Consequently, it is sufficient to find out from which one of the 6 available FIR write layouts we can perform the fastest transposition to one of the 2 available beam former read layouts. Figure 7.3 shows that the optimal pipeline should use STPC, SPTC or TSPC for FIR write layout and SCTP for beam former read layout.

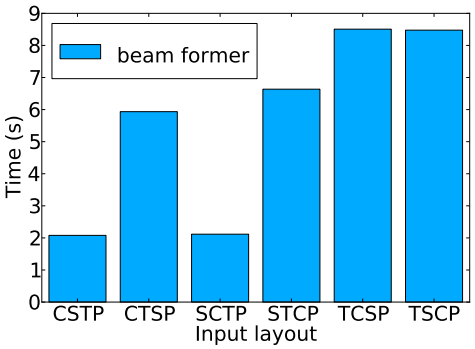
To see how much time we saved by performing the auto-tuning we compared the tuned pipeline to the pipeline with default layouts. The first comparison (Figure 7.4a) hardly shows any performance improvement. This is caused by the very slow and inefficient 3rd party FFT implementation. Therefore, we performed a second comparison, without the FFT (Figure 7.4b), which yields a performance



(a) FIR write layout tuning.



(b) FFT layout tuning.



(c) Beam former read layout tuning.

Figure 7.2: Separate kernel layout tuning for Pipeline 2

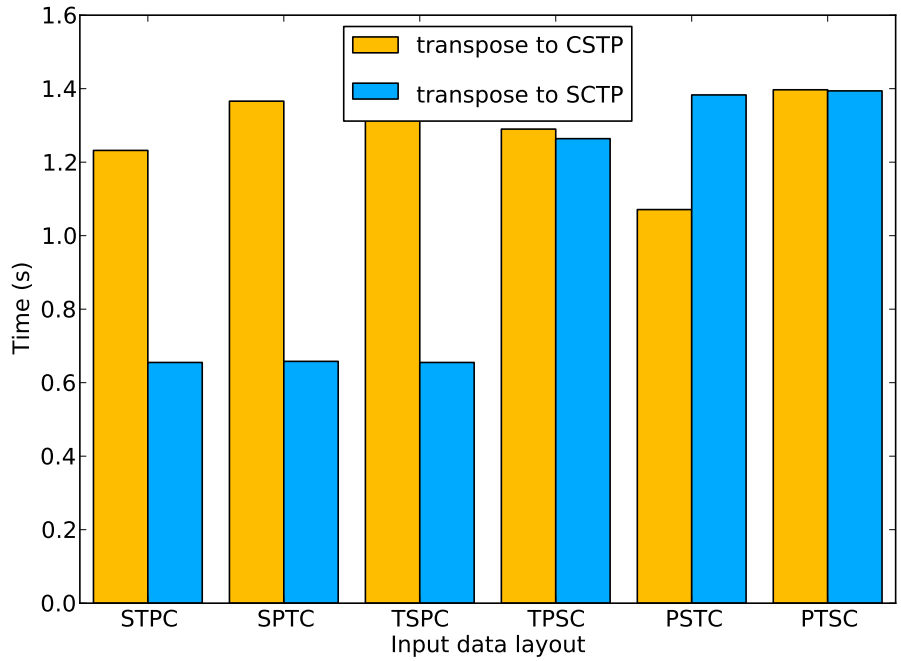
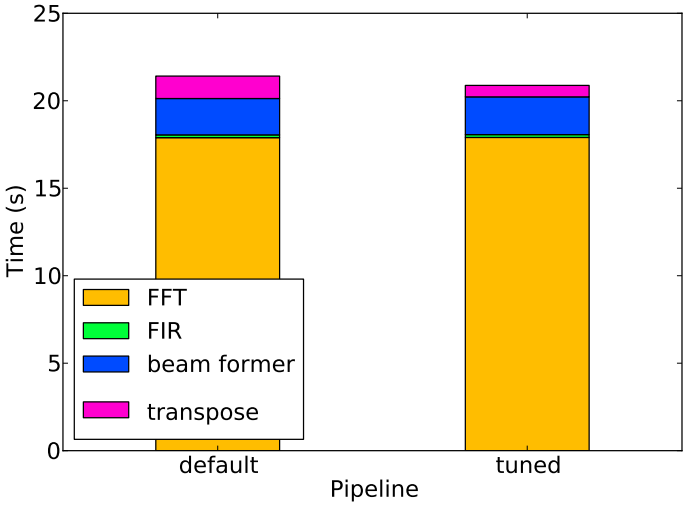


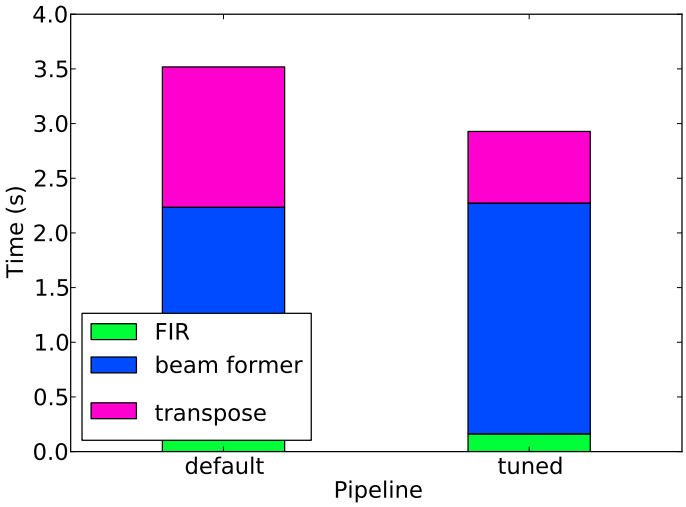
Figure 7.3: Transpose kernel layout tuning. The read layouts (x axis) are the 6 best performing write layouts from Figure 7.2a. The write layouts (series) are the 2 best performing read layouts from Figure 7.2c

comparison	with FFT		without FFT	
pipeline	default	tuned	default	tuned
FIR	0.158	0.162	0.158	0.162
FFT	17.891	17.903	-	-
transpose	1.283	0.656	1.283	0.655
beam former	2.079	2.157	2.077	2.111
total	21.411	20.878	3.518	2.928
improvement	-	**2.5%**	-	**20.1%**

Table 7.1: Comparison of the default and tuned Pipeline 1.



(a) with FFT



(b) without FFT

Figure 7.4: Comparison of the default and tuned Pipeline 1 (the kernels are not in the order in which they appear in the pipeline).

improvement of 20%. The exact times of both comparisons are presented in Table 7.1.

In conclusion, Table 7.1 proves that by performing pipeline layout tuning we were able to employ suboptimal layouts to save transposition time and consequently improve the overall pipeline performance. Currently, having a slow FFT implementation the performance improvement is subtle. Nevertheless, by improving the FFT implementation, we can increase the performance by roughly 20%.

7.5 Tuning Pipeline 2

Since Pipeline 2 is an extension of Pipeline 1 (Figure 7.1b), we used the results from the previous section and only tuned the new part of the pipeline consisting of the beam former and the correlator. In other words, we obtained the beam former read layout, optimal for the old part, separately from the beam former write layout optimal for the new part. What is more, we claim that these layouts are optimal for entire Pipeline 2 and not just for the parts of the pipeline within which they were discovered. This approach could potentially lead to a suboptimal solution in the following situation. Let us assume that we tuned the old part and the new part of the pipeline separately and obtained beam former layouts R_{opt} read layout optimal for the old part and W_{opt} write layout optimal for the new part. Furthermore, let us suppose that when we tuned the old part, the beam former write layout was fixed at a layout W_{opt} and when we tuned the new part, the beam former read layout was fixed at a layout R_{opt}. Now, there is a possibility that there is a different write layout W_{alt} which performs suboptimally within the new part of the pipeline, but to which exists an read layout R_{alt} different from R_{opt} that causes the old part to perform better than under R_{opt} fixed at W_{opt}. To show that this can not happen we verified that for any write layout the ranking (given by performance of the kernel) of the read layouts stays the same (Figure 7.5). We call this kernel property *read-write independence*. The *read-write independence* allows us to tune the new part of the pipeline separately from the old part.

Tuning only the new part of the pipeline means exploring the 6 beam former write layouts together with the 6 correlator read layouts. We again first tuned the layouts separately (Figure 7.6) and made the following observations:

- The beam former write layouts do not vary too much in performance (less than 0.5 second difference). As a result, we have to use all beam former write layouts.

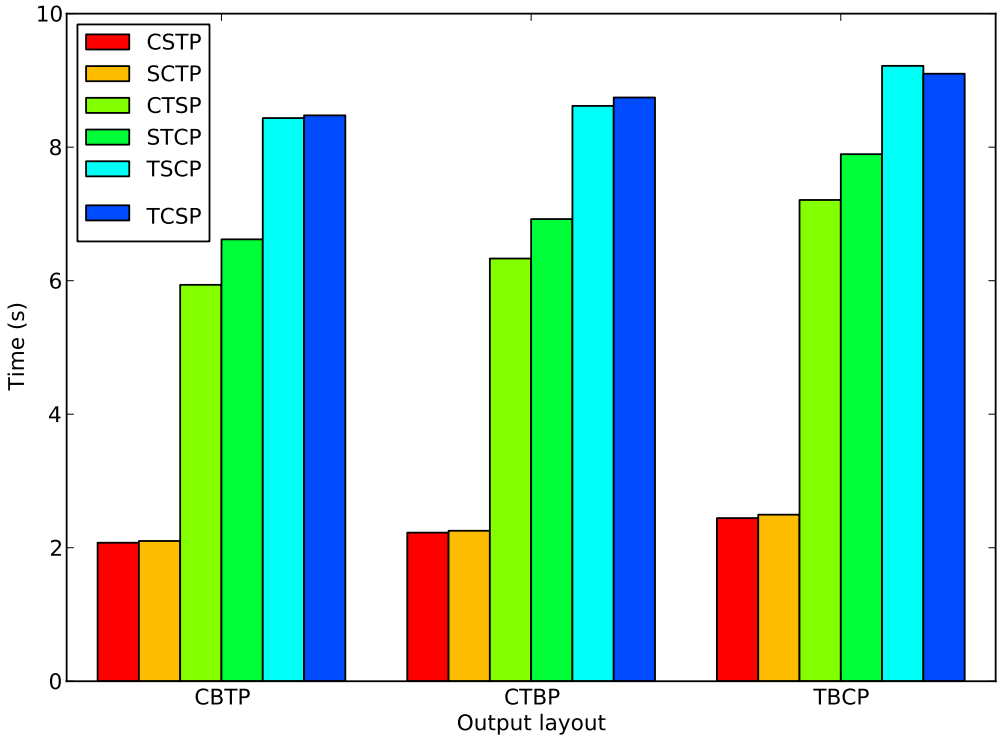


Figure 7.5: Beam former performance under various read and write layouts. Shows all read layouts but only three write layouts (the 1st, 3rd and 6th in terms of performance). The other write layouts look similarly to those shown. The write layout changes with the x axis. The read layout changes with bar colors.

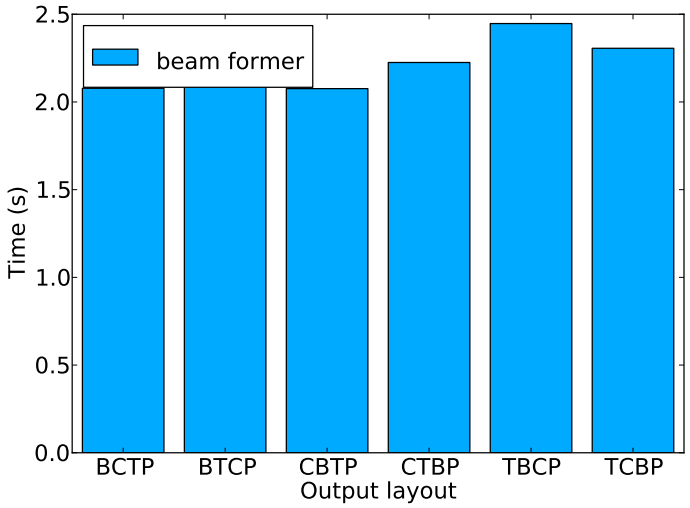
comparison	with FFT		without FFT	
pipeline	default	tuned	default	tuned
FIR	0.158	0.162	0.158	0.162
FFT	17.907	17.938	-	-
transpose	1.283	0.656	1.283	0.655
beam former	2.079	2.28	2.075	2.252
transpose	0.643	-	0.643	-
correlator	4.394	3.29	4.396	3.288
total	26.464	24.325	8.553	6.357
improvement	-	**8.8%**	-	**34.5%**

Table 7.2: Comparison of the default and tuned Pipeline 2 on GPU.

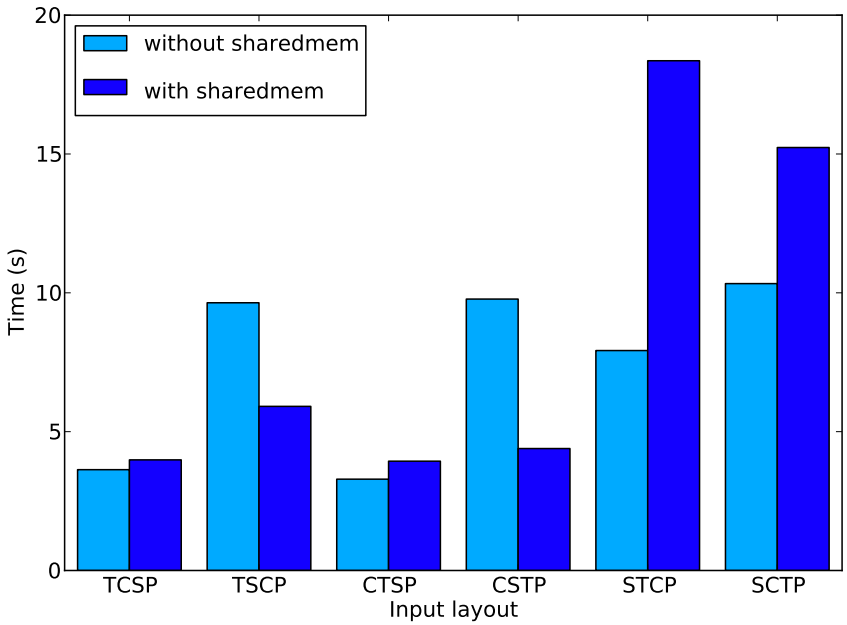
- By disabling the usage of the shared memory in the correlator kernel we were able to get a better performance than with the usage of the shared memory. We achieved this with read layouts with better data locality where the station is on the 3rd place in the layout. Therefore, we attribute the performance improvement to a better cache use. We further use the kernel without the shared memory.
- The 2 best correlator read layouts are significantly better than the rest (by more than 6 seconds). Thus, we can limit ourselves to only these 2 layouts.

Figure 7.7 shows the tuning of the final $6 \times 2 = 12$ layout combinations within the entire pipeline. We left out the FFT filter on purpose, so that its unreasonably long running time does not obscure our results. The best performance is achieved with the beam former write and correlator read layout of CTBP. That means that we can simply remove the transposition kernel between the beam former and correlator to save further time. The comparison between the tuned and default pipeline promises a performance gain of up to 34% (Table 7.2).

All in all, by applying auto-tuning we were able to significantly improve the performance of Pipeline 2. The improvement was achieved by finding a better correlator layout than the default one. The better layout even rendered the transposition between beam former and correlator unnecessary and hence saved further time.



(a) Beam former write layout tuning.



(b) Correlator read layout tuning.

Figure 7.6: Separate kernel layout tuning for Pipeline 2.

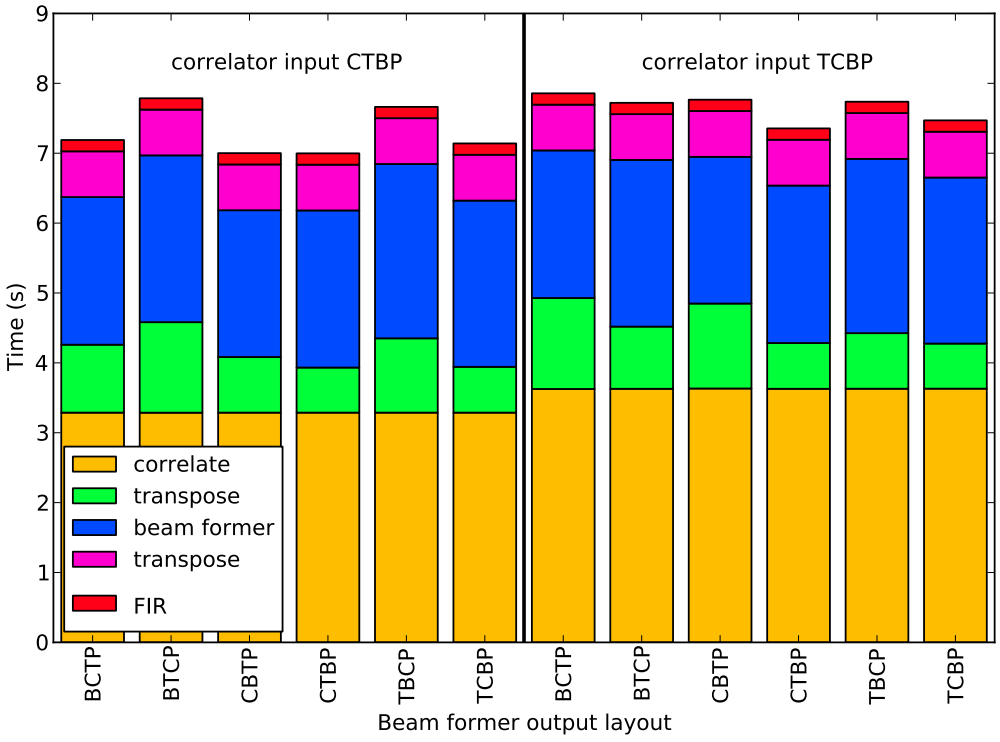


Figure 7.7: Pipeline 2 exploration of the remaining 12 layout combinations.

7.6 Pipeline tuning on CPU

For completeness, we also include the results of pipeline tuning on a CPU (Intel Xeon E5620). The procedure is very similar to the one followed in previous two sections (7.4, 7.5). We first tuned each kernel separately and Afterward based on the results we pruned the search space. Since we tuned Pipeline 2, we verified and confirmed that the *read-write independence* property holds for the beam former also on CPU. Afterward, we were able to tune Pipeline 1 and the new part of Pipeline 2 separately and combine the results. To avoid unnecessary repetition, we only discuss the differences and final results.

Solely for this section we changed our experiment setup to better accommodate CPU. Since the CPU is slower than GPU we only used 10 instead of 100 iterations. Furthermore, we used a beam block of the same size as number of beams, i.e. 100.

When performing the tuning of the FIR write layout, we encountered a great variance between multiple runs. Figure 7.8 shows the variance of 100 FIR write layout tuning experiments. Since we were not able to determine the source of the variance in a reasonable amount of time, we left it for future work and in the further experiments we worked with the mean value (red line in Figure 7.8). We use the mean value also with FFT where we experienced a variance of 1 second dispersed around 5 seconds of total FFT running time. The other kernels (transpose, beam former and correlator) were reasonably stable.

The outcome of the tuning is presented in Table 7.3 which compares the tuned and the default pipeline. The tuned pipeline promises a performance gain of 12%. Most of the gain comes from using CBTP beam former write layout which gives a better beam former performance and also allows us to skip the transpose kernel between beam former and correlator. The performance gain could be larger if we had a better FFT implementation and CPU optimized beam former such as in [32].

7.7 Tuning time

To find out the approximate time it would take to explore all the 864 layout combinations from Pipeline 1 we need to choose an average performing combination. For that purpose, we choose for each kernel a layout closest to the kernel's average. Specifically, it is TCPS FIR write layout and CTSP beam former read layout. We see from Table 7.4 that the total time to run the pipeline is more than 9.7 seconds. That means that the 864 possibilities of Pipeline 1 would take us 2.3 hours

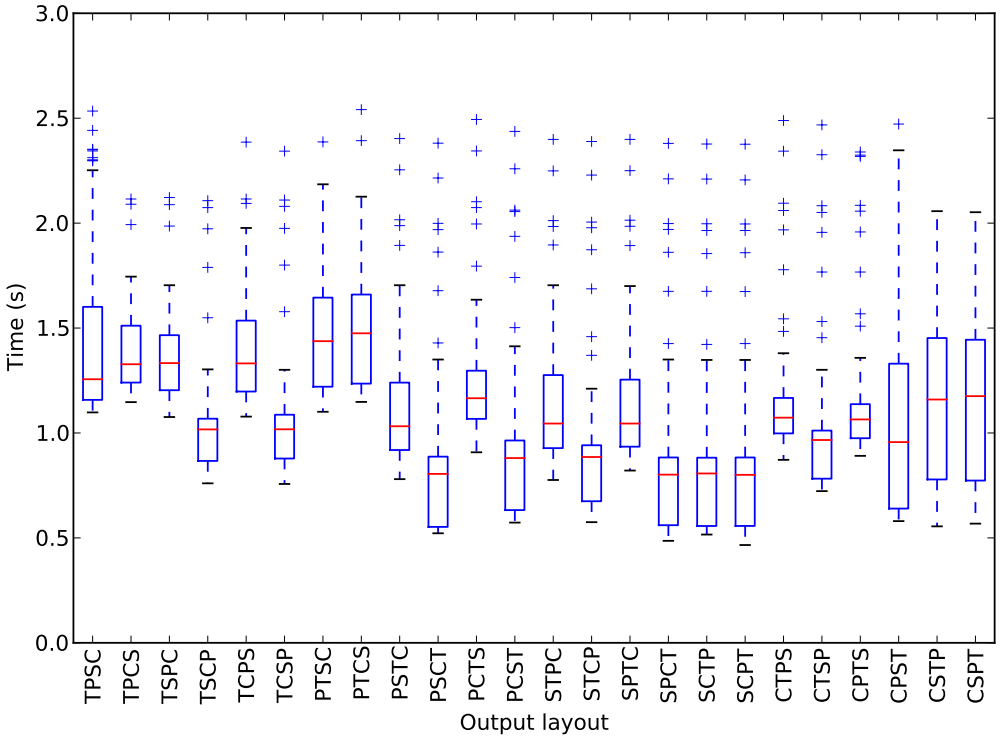


Figure 7.8: Fir write layout tuning performed on Intel Xeon. We performed the experiment 100 times and experienced a great variance among the runs; hence, the box plot representation.

pipeline	default	tuned
FIR	1.045	1.256
FFT	4.841	4.841
transpose	0.954	0.52
beam former	7.543	7.045
transpose	0.799	-
correlator	2.836	2.819
total	18.229	16.27
improvement	-	**12%**

Table 7.3: Comparison of the default and tuned Pipeline 2 on CPU.

	Pipeline 1		Pipeline 2	
	seconds	percents	seconds	percents
kernel init	1.487	15.26	1.25	8.04
kernel build	0.21	2.15	0.949	6.1
kernel run	8.042	82.5	13.335	85.78
total	**9.748**	-	**15.546**	-

Table 7.4: Time of an average pipeline execution. The experiments were performed without the slow and imperfect FFT filter.

to explore. 2.3 hours is too slow for experiments that need to be auto-tuned immediately. However, it is acceptable for the experiments which can be auto-tuned overnight.

Taking a look at the longer Pipeline 2, using the TCBP layout for the correlator, we get 15.5 seconds for the entire pipeline run (Table 7.4). Multiplying that by the 31104 layout combinations we get a tuning time of more than 5.5 days. Such a long tuning time is under any circumstances unacceptable. As a consequence, the search space pruning is inevitable.

To see how much time we saved by performing the search space pruning, we need to calculate how many layout combinations did we explore. We first explored the layout of each kernel separately. That gives us 48 layouts total (24 FIR filter write layouts plus 6 layouts of the FFT plus 6 beam former read layouts plus 6 beam former write layouts and 6 correlator read layouts). Additionally, we explored further 12 layout combinations in Pipeline 1 and 12 more in Pipeline 2. This gives us 60 combinations for Pipeline 1 and 72 combinations for Pipeline 2. For simplicity let us count the time to run one layout exploration as equal to the previously discovered average pipeline run (9.7 seconds). Finally, the time it took us to tune Pipeline 1 was 60×9.7 seconds $= 9.7$ minutes which is 14 times less than the 2.3 hours it would take us to explore all combinations. The time it took us to tune Pipeline 2 was 72×9.7 seconds $= 11.6$ minutes which is 682 times less than the 5.5 days of exploring all combinations.

7.8 Conclusions & recommendations

Based on our results from the full pipeline auto-tuning, we make 3 recommendations:

- Spending time on tuning the entire pipeline rather than just the individual kernels is worth the effort. In our particular case, we were able to improve the performance of Pipeline 2 by 34%.
- It is desirable to first tune the read and write layouts of each kernel separately to gain a better understanding of the kernel which helps us to prune the search space. By performing separate layout tuning we were able to cut the number of explored combinations in Pipeline 1 from 864 to 60 and consequently speed up the tuning process 14 times.
- When performing a data layout tuning in larger pipelines an important kernel property to look for is the read-write independence, which allows to split a large pipeline into smaller pipelines and tune them separately. In our Pipeline 2 we were able to split the original 31104 layout combinations to only $864 + 36$; thus, reducing the tuning time from 5.5 days to a little more than 2.3 hours.

Finally, we note that the lessons learned in this chapter, and the follow up recommendations, are of general nature and they apply to any pipeline with multidimensional data (i.e. not just to the LOFAR pipeline).

Chapter 8

Conclusions

In this chapter we conclude our work with the answers to the four questions from our introduction (at the end of Chapter 1). Additionally, we discuss ideas for future work.

8.1 Answers

At the beginning of our work we identified the need to have the LOFAR pipeline as fast as possible, so that we can perform larger observations. We asked ourselves: **How much performance can we gain by auto-tuning the LOFAR pipeline?** Step by step, we came to a conclusion that we can **gain up to 34%** (Section 7.8). To get the answer we faced and solved the following challenges:

- **How to tune a single kernel**
  To tune a single kernel we first identified the parameters with the greatest performance impact. Afterwards, we constructed a tunable kernel which allowed us to explore all possible values of the parameters. For example, in the case of the correlator we implemented a kernel that could explore all the different cell shapes and sizes. Once we had a tunable kernel we explored the parameters' values. Having several parameters and large range of values for each parameter, we identified a need for search space pruning. First of all, we tuned each parameter separately and discarded the values yielding significantly low performance. Secondly, we performed a combined tuning of all parameters that correlated in performance. In the end we were able

to reach the same performance as the hand-optimized kernel. Summing up, any kernel can by auto-tuned by following three logical steps: identify parameters to tune; design tunable kernel; explore the parameters' values and remember the best ones.

- **How to efficiently connect kernels**
  We examined two possibilities to connect two kernels with different data layouts. The first possibility is for the two kernels to agree on a common layout to use (common write & read). This option can be used easily with kernels that can adjust their layout. However, it is often not usable with 3rd party kernels which can only use a fixed data layout. The second possibility is to put a special transposition kernel between the two kernels that we want to connect. The use of transposition kernel is a very flexible solution that can handle even 3rd party libraries. Its disadvantage is that it doubles the number of accesses to the generally slow global memory. Nevertheless, we showed that there are cases in which the transposition kernel can outperform the common write & read. To sum up, in an application consisting of two or more kernels working upon the same multidimensional data we recommend experimenting with transposition kernel to achieve optimal performance.

- **How to tune an entire pipeline**
  While tuning the entire pipeline we followed a generic process similar to the one followed when we tuned a single kernel. First of all, we identified the data layout as the crucial parameter influencing the performance of the pipeline. Consequently, in each of our kernels (the FIR filter, beam former and correlator), we implemented the ability to read and write the data in any format. Afterwards, we set up the experiments for pipeline data layout tuning and again we recognized the need for search space pruning. Therefore, we tuned the layout of each kernel separately and discarded values of low performance. Furthermore, we found an important property (read-write independence) of beam former kernel which allowed us to split the pipeline into two pipelines and tune them separately; thus, the number of explored combinations was reduced radically. We conclude that the data layout tuning in a pipeline consisting of several kernels has a great potential to increase the overall application performance. However, to make the tuning feasible, search space pruning is inevitable.

All in all, the methodology we applied to auto-tune the LOFAR pipeline helped us to significantly improve the performance. What is more, this methodology is not limited to the LOFAR pipeline only and can be applied to any generic pipeline.

8.2 Future work

As far as the auto-tuning is concerned, there are several directions in which our work can be extended in the future:

- Apply the methodology to other pipelines.
- Automate the process of search space pruning. When tuning a combination of parameters the tuning framework could perform separate parameter tuning first and automatically discard the values of low performance based on pre-set threshold.
- Further improve the performance of the transposition kernel. Alternatively, also explore in-place transpositions.

Considering the overall performance of the LOFAR pipeline, further steps need to be taken in the future to successfully replace the currently used Blue Gene supercomputer. First of all, to match the throughput of Blue Gene a distribution of the computation across several GPUs is required. Secondly, there is a need for much faster FFT than the default JavaCL FFT implementation.

Bibliography

[1] Aparapi: Java/OpenCL library from AMD. http://code.google.com/p/aparapi.

[2] ASTRON: stations. http://www.astron.nl/radio-observatory/astronomers/lofar-astronomers.

[3] IBIS/IPL Java library for distributed computing. http://www.cs.vu.nl/ibis/ipl.html.

[4] JavaCL: open-source Java/OpenCL library from Olivier Chafik. http://code.google.com/p/javacl.

[5] JOCL: open-source Java/OpenCL library. http://www.jocl.org.

[6] MeerKAT. http://www.ska.ac.za/meerkat.

[7] OpenCL standard. http://en.wikipedia.org/wiki/OpenCL.

[8] Square Kilometer Array. http://www.skatelescope.org/about.

[9] F. Agakov, E. Bonilla, J.Cavazos, B.Franke, G. Fursin, M.F.P. O'Boyle, J. Thomson, M. Toussaint, and C.K.I. Williams. Using machine learning to focus iterative optimization. *Code Generation and Optimization*, 2006.

[10] Jason Ansel, Maciej Pacula, Yee Lok Wong, Cy Chan, Marek Olszewski, Una-May O'Reilly, and Saman Amarasinghe. Siblingrivalry: online autotuning through local competitions. *Proceedings of the 2012 international conference on Compilers, architectures and synthesis for embedded systems*, 2012.

[11] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, and Katherine A. Yelick. The

landscape of parallel computing research: A view from berkeley. Technical report, EECS Department, University of California, Berkeley, Dec 2006.

[12] Jong-Ho Byun, Richard Lin, Katherine A. Yelick, and James Demmel. Autotuning sparse matrix-vector multiplication for multicore. Technical report, EECS Department, University of California, Berkeley, Nov 2012.

[13] Shuai Che, Jeremy W. Sheaffer, and Kevin Skadron. Dymaxion: optimizing memory access patterns for heterogeneous systems. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 13:1–13:11, 2011.

[14] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on gpus. *SIGPLAN Not.*, 45(5):115–126, January 2010.

[15] CSIRO. Australian Square Kilometre Array Pathfinder Fast Facts. http://www.atnf.csiro.au/projects/askap/ASKAP_Overview.pdf.

[16] M. De Vos. Lofar: the first of a new generation of radio telescopes. In *Acoustics, Speech, and Signal Processing, 2005. Proceedings. (ICASSP '05). IEEE International Conference on*, volume 5, pages v/865 – v/868 Vol. 5, march 2005.

[17] M. De Vos, A.W. Gunst, and R. Nijboer. The lofar telescope: System architecture and signal processing. *Proceedings of the IEEE*, 97(8):1431 –1437, aug. 2009.

[18] A.T. Deller, S.J. Tingay, M. Bailes, and West. c. Difx: A software correlator for very long baseline interferometry using multi-processor computing environments. *arXiv:astro-ph/0702141v1*, 2007.

[19] C.H.Q. Ding. An optimal index reshuffle algorithm for multidimensional arrays and its applications for parallel architectures. *IEEE Transactions on Parallel and Distributed Systems*, 12, 2001.

[20] M. Frigo and S.G. Johnson. The design and implementation of fftw3. *Proceedings of the IEEE*, 93(2):216 –231, feb. 2005.

[21] MB Giles, GR Mudalige, B. Spencer, C. Bertolli, and I. Reguly. Designing op2 for gpu architectures. *Journal of Parallel and Distributed Computing*, 2012.

[22] S. Kamil, Cy Chan, L. Oliker, J. Shalf, and S. Williams. An auto-tuning framework for parallel multicore stencil computations. In *Parallel Distributed Processing (IPDPS)*, pages 1 –12, april 2010.

[23] NVIDIA. Opencl programming guide for the cuda architecture. http://www.nvidia.com/content/cudazone/download/OpenCL/NVIDIA_OpenCL_ProgrammingGuide.pdf, 2009.

[24] NVIDIA. OpenCL Best Practices Guide. http://developer.download.nvidia.com/compute/DevZone/docs/html/OpenCL/doc/OpenCL_Best_Practices_Guide.pdf, 2011.

[25] NVIDIA. Cufft library. http://docs.nvidia.com/cuda/pdf/CUDA_CUFFT_Users_Guide.pdf, 2012.

[26] NVIDIA, Matt Pharr, and Randima Fernando. Gpu gems 2: Programming techniques for high-performance graphics and general-purpose computation, 2005.

[27] Markus Püschel, José M. F. Moura, Jeremy Johnson, David Padua, Manuela Veloso, Bryan Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W. Johnson, and Nicholas Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232–275, 2005.

[28] John W. Romein, P. Chris Broekema, Jan David Mol, and Rob V. van Nieuwpoort. The LOFAR Correlator: Implementation and Performance Analysis. In *ACM Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, pages 169–178, Bangalore, India, January 2010.

[29] J. Roy, Y. Gupta, U. Pen, J.B. Peterson, S. Kudale, and J. Kodilkar. A real-time software backend for the gmrt. *arXiv:0910.1517v2 [astro-ph.IM]*, 2010.

[30] G. Ruetsch and M. Paulius. Optimizing matrix transpose in cuda. http://docs.nvidia.com/cuda/samples/6_Advanced/transpose/doc/MatrixTranspose.pdf, 2009.

[31] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-il: An instrumentation language for auto-tuning parallel applications. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*. Springer-Verlag, 2009.

[32] Alessio Sclocco, Ana Lucia Varbanescu, Jan David Mol, and Rob V. van Nieuwpoort. Radio astronomy beam forming on many-core architectures. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, 2012.

[33] B. Spencer. A general auto-tuning framework for software performance optimisation, 2011.

[34] I-Jui Sung, G.D. Liu, and W.-M.W. Hwu. Dl: A data layout transformation system for heterogeneous computing. In *Innovative Parallel Computing (InPar), 2012*, pages 1 –11, may 2012.

[35] I-Jui Sung, John A. Stratton, and Wen-Mei W. Hwu. Data layout transformation exploiting memory-level parallelism in structured grid many-core applications. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, pages 513–522, 2010.

[36] Victor Pankratius Thomas Karcher. Run-time automatic performance tuning for multicore applications. *Euro-Par 2011 Parallel Processing*, 2011.

[37] A. Tiwari, Chun Chen, J. Chame, M. Hall, and J.K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *Parallel Distributed Processing*, may 2009.

[38] Karel van der Veldt, Rob van Niewpoort, Ana Lucia Varbanescu, and Chris Jesshope. A polyphase filter for gpus and multi-core processors. *Astro-HPC '12 Proceedings of the 2012 workshop on High-Performance Computing for Astronomy Date*, 2012.

[39] Rob V. van Nieuwpoort and John W. Romein. Correlating Radio Astronomy Signals with Many-Core Hardware. *Springer International Journal of Parallel Programming*, 39, 2011.

[40] R. Clint Whaley and Antoine Petitet. Minimizing development and maintenance costs in supporting persistently optimized blas. *Softw. Pract. Exper.*, 35(2):101–121, February 2005.

[41] Samuel Williams, Kaushik Datta, Jonathan Carter, Leonid Oliker, John Shalf, Katherine Yelick, and David Bailey. Peri- auto-tuning memory intensive kernels for multicore, 2008.

[42] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4), April 2009.

[43] K. Yotov, X. Li, G. Ren, M.J.S. Garzaran, D. Padua, K. Pingali, and P. Stodghill. Is search really necessary to generate high-performance blas? *Proceedings of the IEEE*, 93(2):358 –386, feb. 2005.