A Polyphase Filter for GPUs and Multi-Core Processors

Karel van der Veldt
Universiteit van Amsterdam
The Netherlands
karel.vd.veldt@uva.nl

Rob van Nieuwpoort
Vrije Universiteit Amsterdam
The Netherlands
r.v.van.nieuwpoort@vu.nl

Ana Lucia Varbanescu
Technische Universiteit Delft
The Netherlands
a.l.varbanescu@tudelft.nl

Chris Jesshope
Universiteit van Amsterdam
The Netherlands
c.r.jesshope@uva.nl

ABSTRACT

Software radio telescopes are a new development in radio astronomy. Rather than using expensive dishes, they form distributed sensor networks of tens of thousands of simple receivers. Signals are processed in software instead of custom-built hardware, taking advantage of the flexibility that software solutions offer. In turn, the data rates are high and the processing requirements challenging. GPUs and multi-core processors are promising devices to provide the required processing power. LOFAR[1], the largest radio telescope, is a prime example of a software radio telescope.
In this paper, we discuss an optimized implementation of the polyphase filter bank used by LOFAR. We compare the following architectures: Intel Core i7, NVIDIA GTX580, ATI HD5870, and MicroGrid[7]. We present a novel way to compute polyphase filters efficiently on GPUs, and also discuss hardware limitations and energy efficiency.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*parallel programming*; D.2.8 [**Software Engineering**]: Metrics—*performance measures*

General Terms

Algorithms, Measurement, Performance

Keywords

LOFAR, Radio Astronomy, Digital Signal Processing, Polyphase Filter, FIR Filter, CUDA, OpenCL, MicroGrid

1. INTRODUCTION

Modern radio telescopes use many separate receivers as building blocks, and combine their signals to form a single large and sensitive instrument. The enormous amounts of data collected are processed mostly in software, in real-time, since the data streams simply are too large to store on disk. Therefore, a scalable solution for processing all this data is needed. For example, the LOFAR radio telescope produces over 100 TB of data daily. If clever solutions can be found for LOFAR, they can also be applied to the future SKA[2] telescope[3], estimated to produce exa-scale data collections *every day*.
In practice, receivers (antennas) are grouped in *stations*. At the station level, signals from the antennas are combined and streamed to the digital signal processing pipeline. One such pipeline is the *imaging pipeline*, used to create images of the sky. The first stage in the imaging pipeline is the polyphase filter (PPF)[12]. The *channelized* data streams that it produces enable better removal of Radio Frequency Interference (RFI), and allows more accurate processing in general. For example, dispersion of the different signal frequencies can be corrected more accurately. A fast PPF allows for more accurate RFI removal, increasing the accuracy of the entire telescope.
The main reason to process radio astronomy data in software rather than custom-built hardware is flexibility: the pipelines can easily be reconfigured and reprogrammed at observation time. However, in supercomputer-based infrastructures such as the Blue Gene/P currently used by LOFAR, the price-to-scale ratio becomes steep in terms of both energy and maintenance costs. Moreover, for the future SKA telescope, we need to scale up the processing with several orders of magnitude, to exascale. A possible alternative to supercomputers is the use of many-core processors, which promise to be cheaper and more energy efficient.
In this paper, we investigate how a PPF can be implemented efficiently in terms of both performance and power consumption. Our investigation covers several many-core architectures: Intel Core i7 920 CPU, NVIDIA GTX580 GPU, ATI Radeon HD5870 GPU, and MicroGrid[7] (a research project by the University of Amsterdam), including different programming models (where applicable). We expect the results of this research to be of high interest for SKA, as it will face the same data processing issues at exa-scale level.
Our main contributions in this work are the parallel solutions for building efficient PPFs on many-core architectures, and GPU-specific optimizations that allowed us to obtain very high performance. Additionally, our PPF is the first "real-world" application written and benchmarked on the

[1]LOw Frequency ARray

[2]Square Kilometer Array

MicroGrid architecture, exposing the programmability and performance abilities of this research architecture. Finally, both the optimizations and the results presented can be used to implement the entire pipeline (or other signal processing kernels) on many-core platforms.

2. RELATED WORK

In this section we discuss other work related to FIR filter and polyphase filter implementations.
In their paper[15], Rob V. van Nieuwpoort and John W. Romein describe their optimized implementation of the LOFAR correlator on various multicore platforms. The best performance is achieved on the IBM Cell/B.E. (full blade), reaching 91% peak performance, compared to 96% on the Blue Gene/P. The Cell/B.E. is also 3.9x more energy efficient than the BG/P.
In 2005, Smirnov and Chiueh describe a GPGPU implementation of a FIR filter using OpenGL [14]. At the time, CUDA and OpenCL did not exist yet.
An implementation of a polyphase filter on the Cell Broadband Engine that is similar to ours was presented by Hamilton in his master's thesis [6]. His results show that the implementation is over 6x more efficient than on a normal processor, depending on the amount of input.
The master's thesis by Pettersson and Wainwright [11] discusses the implementation and performance of FIR filters on CUDA and OpenCL. They achieve good performance on CUDA, but they do not provide much detail on the actual implementation. Their FIR filter parameters also differ from ours.
The SPIRAL Project [13] researches automatic code generation for the development and optimization of DSP algorithms and other numerical kernels, including FIR filters and FFTs. The generated code outperforms existing, handwritten libraries, but is not very flexible and there is no GPU code generation.
Overall, we believe that although signal processing in general and FIR filters in particular are of interest to the many-core community, this is the first thorough study of FIR filters using so many platforms, programming models, and performance metrics.

3. SIGNAL PROCESSING BACKGROUND

In this section we give a short description of the signal processing concepts required to understand polyphase filters.

3.1 Signals

A *signal* is defined as any physical quantity that varies with time, space, or other independent variable(s) [12]. A signal can be mathematically described as a function of one or more independent variables. In this work, we are only interested in discrete signals.
Discrete signals can be obtained by sampling at (usually) equally spaced intervals from an analog signal source. In our case, LOFAR antennas sample discrete, complex-valued samples, using sampling frequencies of 160 or 200 MHz.

3.2 FIR filter

A Finite Impulse Response (FIR) filter multiplies a finite number of recent input signals (impulses) relative to a given discrete time by coefficients (impulse responses) and accumulates the results. It can be described mathematically as $y(n) = \sum_{i=0}^{N} c_i x(n-i)$, where:

- $y(n)$ is the output signal at discrete time n.
- $x(n)$ is the input signal at discrete time n.
- c_i are the coefficients, also called *weights*.
- N is the number of recent signals to consider, called the *filter order*. The terms on the right-hand side of the equation are called *taps*. An Nth order FIR filter has $N+1$ taps.

A FIR filter must remember its last N input samples, which are stored in what is called the *delay line*. One can design a FIR filter by carefully choosing the filter order and coefficients such that the system has specific characteristics. For the purpose of our work, the values of the coefficients are irrelevant as they do not affect the implementation. While generally it is possible to reduce the complexity of FIR filters by strength reduction [10], this is not feasible for us as it involves designing a specific FIR filter for a specific set of coefficients. In LOFAR there are hundreds of different FIR configurations, all of which can be changed at any time.

3.3 Discrete Fourier Transform

A Fourier transform splits a sequence of input signals into a sequence of frequencies. In doing so it transforms the input from the *time domain* to the *frequency domain*. It can be compared to how a prism splits white light into separate light beams of a single frequency.
A DFT operates on discrete signals and can be described mathematically as $f_k = \sum_{n=0}^{N-1} x(n) e^{-i\frac{2\pi}{N}nk}$, where:

- $x(n)$ is an input signal; there are N input signals.
- f_k is the kth frequency and is a complex number, $k = 0, 1, 2, ..., N-1$.

The complexity of this algorithm is $O(N^2)$, since computing any of the N frequencies requires iterating over N inputs. DFTs are not used directly in practice, because there are better algorithms known as *Fast Fourier Transforms* (FFT) which have a complexity of only $O(N \log_2(N))$ [4].

3.4 Polyphase filter

Polyphase filters are used by LOFAR to *channelize* input streams and reduce interference. They split an input sequence into N subsequences of M samples, where each subsequent input signal is the input to one of M FIR filters (or *channels*). This can be described mathematically as $y_m(n) = \sum_{i=0}^{N} c_i x((n-i)M + m)$, where:

- N is the number of recent samples to consider (the filter order).
- M is the number of FIR filters (channels).
- $y_m(n)$ is the nth output signal of the mth FIR filter, $m = 0, 1, 2, ..., M-1$.

The M outputs $y_m(n)$ are used as inputs to a DFT as described in the previous subsection. The output of the DFT is the output of the polyphase filter.

4. THE LOFAR POLYPHASE FILTER

In this section we present the implementation details common to all architectures we implemented the polyphase filter on, and how we measure performance. We focus on the implementation of the FIR filter, as we use third-party FFT libraries when possible.

4.1 Polyphase filter

In the LOFAR system, receivers are grouped into *stations*. As all stations are completely independent, we explain how the polyphase filter works for a single station.

A station has $N_{channels}$ channels, which each have two polarizations (X and Y). Polarizations are separate interleaved data streams that share the same FIR coefficients. There are a total of $2 \times N_{stations} \times N_{channels}$ polyphase filters. Each station combines the samples of its receivers and streams it to the LOFAR pipeline.

Samples from the stream are 4, 8, or 16-bit interleaved complex integers, which the polyphase filter first converts to 32-bit floating point. The FIR coefficients are 32-bit floating point real numbers. There is a coefficient for every channel and tap combination, but all stations and polarizations share the same coefficients. The FIR delay line can be seen as a bounded FIFO buffer. When a new sample is processed it is stored in the front of the buffer, all other samples shift to the next tap, and the last sample is discarded. After all FIRs of a given polarization have processed a sample, the FFT is computed. There are $2 \times N_{stations}$ FFTs of $N_{channels}$ length.

In our implementation the input samples are read from an *input array*, and the result is stored in an *output array*, which are large enough to store a number of samples described above for the $N_{stations}$ we want to process. We also use a *delay line array* and a *coefficients array*.

4.2 Measuring performance

In this section we explain how we measure the performance of our kernels.

4.2.1 *Floating point operations*

Computing the output of a FIR filter requires a number of multiply-add operations. There are N_{taps} complex samples in the delay line. Each sample is multiplied by a real coefficient and these results are summed. This requires $2N_{taps}$ floating point multiplications and $2(N_{taps}-1)$ floating point additions. The total amount of FLOPs per FIR filter is thus $2 + 4(N_{taps} - 1)$.

Since we use third-party FFT libraries we do not know the exact number of FLOPs for the FFT, but it can be approximated as $5N_{channels} \log_2(N_{channels})$ [9]. LOFAR only uses power of two FFTs, because those can be computed most efficiently.

4.2.2 *Memory traffic*

Computing the output of a FIR filter requires the following memory loads and stores:

- Read one (2 x 4 bit), (2 x 8 bit) or (2 x 16 bit) input sample, which is converted to a (2 x 32 bit) floating point sample. Note that for simplicity of the calculations we need to make we assume (2 x 16 bit) samples.
- Read $(N_{taps} - 1)$ (2 x 32 bit) samples from the delay line.
- Read N_{taps} 32 bit coefficients.
- Write one (2 x 32 bit) output.
- Write one (2 x 32 bit) sample to the delay line.

So, the total amount of memory traffic for one FIR filter is $4 + 8(N_{taps} - 1) + 4N_{taps} + 8 + 8 = (12N_{taps} - 4) + 16 = 12N_{taps} + 12$ bytes.

One FFT has in total $4N_{channels}$ [9] complex floating point inputs and outputs, so the amount of memory traffic is $8 \times 4N_{channels} = 32N_{channels}$ bytes.

4.2.3 *Peak performance*

We use the Roofline model[16] to determine the maximum attainable performance of our implementation on a given architecture:

$perf_{max} = \min(perf_{peak}, MemoryBandwidth \times AI)$, where:

- $perf_{max}$ is the maximum attainable floating point performance of our implementation on the given architecture (GFLOP/s).
- $perf_{peak}$ is the theoretical peak floating point performance of the architecture (GFLOP/s).
- $MemoryBandwidth$ is the peak memory bandwidth of the architecture (GB/s).
- AI is the *arithmetic intensity* of the implementation, which is defined as the number of FLOPs per byte of memory traffic. The AI of the polyphase filter is given in the following subsection.

Using the Roofline model we can determine whether our kernels are bounded by computational power of the processor or by the memory bandwidth. If the measured performance of a kernel is lower than $perf_{max}$, it is memory bound. Otherwise, it is compute bound. Note that because the Roofline model does not take all possible optimizations (such as caching) into account, there are cases when the measured performance is higher than $perf_{max}$.

4.2.4 *Arithmetic intensity*

To use the Roofline model, we must determine the arithmetic intensity of our kernel. Arithmetic intensity is defined as the number of FLOPs per byte of memory traffic, so we need to calculate both. We calculate the AI of the FIR filter and FFT separately.

$$\begin{aligned}
FLOP_{fir} &= 2 + 4(N_{taps} - 1) \\
BytesAccessed_{fir} &= 12N_{taps} + 12 \\
AI_{fir} &= FLOP_{fir}/BytesAccessed_{fir} \\
FLOP_{fft} &= 5N_{channels} \log_2(N_{channels}) \\
BytesAccessed_{fft} &= 32N_{channels} \\
AI_{fft} &= FLOP_{fft}/BytesAccessed_{fft}
\end{aligned} \tag{1}$$

Note that for some of our implementations there are certain optimizations which improve the AI, as explained in Sec. 5.

4.3 Parameters and metrics

We made test programs to measure the performance of our kernels based on general and implementation-specific parameters. The general parameters are: sample size, $N_{stations}$, $N_{channels}$, N_{taps}, and the number of input samples per channel N_{runs} (in other words the number of times to run the polyphase filter). We call the act of starting the kernel to process a sample a *run*, and every run is performed in lockstep by all polyphase filters. Implementation-specific parameters include enabled optimizations (determined at compilation time) and additional command line parameters, for example the number of threads in the CPU implementation. We kept N_{runs} at 10000, but varied all the other parameters. The following metrics are used to evaluate performance: execution time in seconds for computing the total number of samples, average time for all channels of all stations to process one input sample, and energy consumption in Watt.

5. ARCHITECTURES

In this section we explain how we optimized the polyphase filter for the following architectures: Intel Core i7 920, NVIDIA GTX580 Fermi, ATI HD5870, and MicroGrid. For comparison, we also have an unoptimized reference implementation for all architectures. Reference implementations are designated with subscript *ref*, and optimized implementations with subscript *opt*.

5.1 Intel Core i7 920

The Core i7 920 is a quad-core running at 2.67 GHz, 32Kb L1 cache, 85 GFLOPs/chip theoretical peak, and the memory bandwidth is 25.6 GB/s. We use the FFTW[5] library for the FFT.

The delay line is implemented as a bounded circular FIFO buffer. On insertion the oldest sample is overwritten, discarding it. Insertion is $O(1)$ as it only requires the start of buffer index to be incremented by $1 \mod N_{taps}$ and the new sample is stored in that location. No copying takes place. To compute the FIR output we iterate over the whole buffer starting at the aforementioned buffer index.

We use a combination of loop unrolling and SSE to optimize iteration and computation. Since polarized samples are stored interleaved they can be loaded into one SSE register in a single SSE instruction, and both polarizations can be computed in parallel. Finally, we use OpenMP to parallelize the stations over a number of threads. We measured with 1, 2, 4, and 8 threads. Not surprisingly 4 threads gave the best performance, as it is equal to the number of cores.

5.1.1 *Maximum performance*

To compute the maximum performance, we need to know the number of FLOPs and bytes accessed per FIR filter and FFT. For the FIR reference implementation and FFT we already know the number of flops and bytes accessed from section 4.2.3. Since we use SSE to compute two polarizations at once, the numbers are computed differently for the optimized implementation:

$$\begin{aligned} FLOP_{fir,ref} &= 2 + 4(N_{taps} - 1) \\ BytesAccessed_{fir,ref} &= 12N_{taps} + 12 \\ FLOP_{fir,opt} &= 4 + 8(N_{taps} - 1) \\ BytesAccessed_{fir,opt} &= 20N_{taps} + 24 \end{aligned} \quad (2)$$

Based on these equations, we can compute the arithmetic intensity and peak performance of the polyphase filter. The performance of the FIR depends on N_{taps}, and the performance on the FFT depends on $N_{channels}$.

The $peak_{max}$ in GFLOP/s for the FIR and FFT are shown in Table 1. The observed performance (see Section 6) is actually much higher, due to the effect of caching.

5.2 NVIDIA GTX580 Fermi

The GTX580 GPU has 512 cores with a clock frequency of 772 MHz divided over 16 symmetric multiprocessors (SM).

N_{taps}	4	8	16	32	64
$AI_{fir,ref}$	0.23	0.28	0.30	0.31	0.33
$AI_{fir,opt}$	0.26	0.33	0.36	0.38	0.39
$perf_{max,fir,ref}$	6.0	7.1	7.8	8.1	8.3
$perf_{max,fir,opt}$	6.9	8.3	9.2	9.7	10.0
$N_{channels}$	64	128	256	512	1024
AI_{fft}	0.94	1.1	1.25	1.4	1.6
$perf_{max,fft}$	24	28	32	36	40

Table 1: The arithmetic intensity and maximum performance of the polyphase filter on the Intel Core i7 920 determined using the Roofline model. $perf_{peak}$ is 85 GFLOP/s and $MemoryBandwidth$ is 25.6 GB/s.

The theoretical peak performance is 1581.1 GFLOP/s per chip. The theoretical peak global memory bandwidth is 192.4 GB/s, and the theoretical peak PCI express bus 2.0 bandwidth is 8 GB/s. Every SM has a register file of 32768 32-bit registers, which is shared between all its cores. We used CUDA 4.1 with CUFFT. We also experimented with the GTX480 using CUDA 3.1 and OpenCL with Apple's FFT library.

The GTX580 has multiple memories with different characteristics, but we only used the global memory for the input, output and delay lines arrays, and the constant memory to store the coefficients. All arrays are arranged in such a way that accesses are coalesced as much as possible. Furthermore, while diverging branches in GPU threads are known to be expensive, our implementation has *no* diverging branches.

In the following subsections we present and analyze a novel approach to FIR filter computation on GPUs using a combination of register heavy threads, aggressive loop unrolling, and batching. These optimizations go hand in hand to make effective use of available resources, and give a very substantial performance boost over a naive implementation.

5.2.1 *Batch processing*

Just as in the CPU implementation (see section 5.1), the FIR delay line is stored in a bounded circular FIFO buffer, but now the buffer is completely loaded into registers, and we only use global memory to store the delay line in between kernel calls. Because of the large number of registers required, a thread computes only a single polarization, and we create $2 \times N_{stations} \times N_{channels}$ FIR filter threads.

Since registers cannot be indexed, we unrolled the FIR loop N_{taps} times using manual register renaming (using C macros) to simulate shifting taps in the delay line without needing to do any copying. The unrolled loop is repeated another N_{taps} times and wrapped in an outer loop. This lets us compute $N_{batches}$ *batches* of N_{taps} samples each within a single kernel call, greatly reducing the total number of memory accesses. The number of samples processed by the kernel is $N_{samples} = N_{batches} \times N_{taps}$. Since the delay line is only read from and written to global memory *once* every $N_{samples}$ samples, the number of bytes accessed is:

$$BytesAccessed_{fir} = 2\frac{8N_{taps}}{N_{samples}} + 4N_{taps} + 12 = \frac{16}{N_{batches}} + 4N_{taps} + 12$$

Now it is clear that, as $N_{batches}$ increases, the factor $\frac{16}{N_{batches}}$ approaches zero, and effectively $BytesAccessed_{fir} \approx 4N_{taps} + 12$, meaning batching masks the memory access latencies that would otherwise be caused by accessing the delay lines from global memory. Since fewer memory accesses are re-

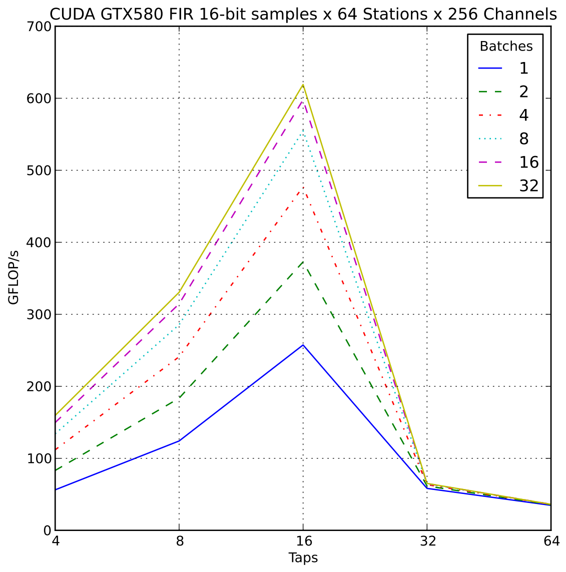


Figure 1: Performance graph showing the impact of the number of taps and batches of the optimized FIR filter without I/O on the GTX580 using CUDA.

quired for the same amount of computation, the arithmetic intensity increases as $N_{batches}$ increases. We measured with $N_{batches} = 1, 2, 4, 8, 16$, and 32, the latter giving the best performance. From the equation above we also know that a larger $N_{batches}$ does not give further performance increase. Table 2 shows the best case arithmetic intensity when $N_{batches} = 32$, and the maximum performance as determined by Roofline. The actual performance is much higher, because of caching [15] and our use of the constant memory which has a higher bandwidth than the global memory.

N_{taps} x 32 batches	4	8	16	32	64
$BytesAccessed_{fir,ref}$	60	108	204	396	780
$BytesAccessed_{fir,opt}$	28	44	76	140	268
AI_{fir}	0.49	0.67	0.81	0.90	0.95
$perf_{max,fir,ref}$	44.9	53.4	58.5	61.2	62.7
$perf_{max,fir,opt}$	96.2	131.2	157.0	173.2	182.3
$N_{channels}$	64	128	256	512	1024
AI_{fft}	0.94	1.1	1.25	1.4	1.6
$perf_{max,fft}$	180.9	211.6	240.5	269.36	307.8

Table 2: The maximum performance of the polyphase filter on the NVIDIA GTX580, *excluding* host-to-device memory transfers. $perf_{peak} = 1581.1$ **GFLOP/s and** $MemoryBandwidth = 192.4$ **GB/s.**

5.2.2 *Occupancy*

Occupancy is a measure of how well the multiprocessor is utilized by a kernel which is based on the number of registers per thread, amount of shared memory per thread (although we do not use shared memory), and the number of threads per block. Best practice guidelines state that it should be as close to 100% as possible. Table 3 shows the occupancy for FIR filters of different lengths, which we computed using the CUDA Occupancy Calculator.
On the GTX580, threads can use a maximum of 63 registers without spilling registers to device memory, and each multiprocessor has 32678 registers to allocate between threads in a warp [1]. Keeping that in mind, the table shows that the 16 taps FIR filter makes near optimal use of the available registers (32256 out of 32768 registers are used) without exceeding the max. registers/thread. This is reflected in the performance measurements shown in Figure 1, as this FIR filter is by far the best performing one. FIR filters with more taps exceed the max. registers/thread and therefore must spill registers, impacting their performance. Moreover, smaller FIR filters have higher occupancy but less performance than the 16 taps FIR filter, because the hardware is sub-optimally utilized.
This shows that higher occupancy does not imply better performance, and to get the best performance one should use as many registers as possible without exceeding the max. register per thread. It also means our FIR filter implementation scales with the max. registers/thread, which is unfortunate as it is a hardware limit we cannot do anything about. As also implied by the table, we need a separate kernel for each N_{taps}, because the number of registers must be hardcoded.

N_{taps}	Registers per thread	Max. threads per block	Total nr. of registers	Occupancy
4	18	512	27684	100%
8	26	512	26624	67%
16	42	256	32256	50%
32	74	128	28416	25%
64	138	32	30912	15%

Table 3: CUDA occupancy on compute ability 2.0. Registers per thread $= 2N_{taps} + 10$**.**

As shown in Table 3, the maximum size of a thread block depends on the number of taps. There is one thread for each channel and polarization in a station, so if $2N_{channels} > MaxThreadsPerBlock$, we must use multiple thread blocks per station. $MaxThreadsPerBlock$ is given in Table 3. However, all thread blocks must have the same size, so we choose $ThreadsPerBlock$ and $BlocksPerStation$ such that:

$$2N_{channels} = ThreadsPerBlock \times BlocksPerStation$$
$$\text{where } ThreadsPerBlock \leq MaxThreadsPerBlock$$

Our implementation computes $ThreadsPerBlock$ and $BlocksPerStation$ automatically, based on the number of channels and taps.
The consequence of this dynamic sizing is that depending on the number of channels, thread blocks may be smaller than optimal, affecting performance (since the occupancy will be lower than shown in Table 3). We strongly recommend choosing $N_{channels}$ such that $ThreadsPerBlock = MaxThreadsPerBlock$.

5.2.3 *I/O transfers*

The input array is *pagelocked* (or *pinned*), *write-combined*, and *mapped into device memory*. This minimizes transfer overhead and the GPU can automatically overlap I/O transfers with computations. We did not apply this to the output array as it is supposed to be reused as input for the following pipeline stage kernel, while the mentioned optimizations only apply to device read-only or write-only data. These optimizations give a substantial I/O performance boost.

5.3 ATI Radeon HD5870

The Radeon HD5870 GPU has 320 *stream cores* running at 850 MHz divided over 20 *compute cores*. Its theoretical peak performance is 2720 GFLOP/s, its peak memory bandwidth is 154 GB/s, and the theoretical peak PCI express bus 2.0 bandwidth is 8 GB/s. ATI uses different terms to describe its GPU architecture, but it is for the most part similar to NVIDIA GPUs. Each stream core has 5 FPUs and its own vector register file. Each register is 4 x 32-bit wide. This is different from the GTX580, where one SM shares its register file between all its cores and registers can only store 1x32-bit values. Each stream core can use at most 1024 registers. The memory architecture is very similar to CUDA, and the same recommendations apply. The HD5870 is programmed using OpenCL.

5.3.1 Implementation

We have two OpenCL implementations. One is a direct port of the CUDA implementation, in which a thread computes one polarization of one channel. In the second (vectorized) implementation a thread computes both polarizations of a channel at once, taking advantage of the vector registers in the same way we applied SSE in the CPU implementation. This means there are half as many threads, but each thread requires twice as many registers. Since two delay lines are accessed and two samples are computed in parallel, but both use the same set of coefficients:

$$BytesAccessed_{fir} = \frac{32}{N_{batches}} + 4N_{taps} + 24$$

And, because both polarizations are computed at once:

$$FLOP_{fir} = 4 + 8(N_{taps} - 1)$$

The OpenCL compiler was unable to compile kernels for 64 taps (it just crashed), so we have no results of that. This is a problem with the compiler, not our code. We also use pagelocked memory to boost I/O performance.

Table 4 shows the maximum performance of the vectorized and non-vectorized reference and optimized implementations.

N_{taps} x 32 batches	4	8	16	32	64
Non-vectorized					
$BytesAccessed_{fir,ref}$	60	108	204	396	780
$BytesAccessed_{fir,opt}$	28	44	76	140	268
AI_{fir}	0.49	0.67	0.81	0.90	0.95
$perf_{max,fir,ref}$	33.9	41.6	46.2	49.3	49.3
$perf_{max,fir,opt}$	75.6	103.8	124.8	138.1	145.7
Vectorized					
$BytesAccessed_{fir,ref}$	108	188	348	668	1308
$BytesAccessed_{fir,opt}$	41	57	89	153	281
AI_{fir}	0.68	1.05	1.39	1.64	1.80
$perf_{max,fir,ref}$	39.0	47.9	53.2	56.8	56.8
$perf_{max,fir,opt}$	105.2	162.1	214.6	253.6	278.4
$N_{channels}$	64	128	256	512	1024
AI_{fft}	0.94	1.1	1.25	1.4	1.6
$perf_{max,fft}$	144.8	169.4	192.5	215.6	246.4

Table 4: The maximum performance of the polyphase filter on the HD5870 (non-vectorized and vectorized), *excluding* host-to-device memory transfers. $perf_{peak}$ = 2720 **GFLOP/s and** $MemoryBandwidth = 154$ **GB/s.**

5.4 MicroGrid

MicroGrid is an NWO (Netherlands Organisation for Scientific Research) funded research project conducted at the University of Amsterdam, aiming to improve the speedup, programmability, power dissipation, scalability and concurrency management of many-core processor architectures [2]. It introduces a new concurrency model called SVP (Self-adaptive Virtual Processor) [7].

We used the MicroGrid simulator to run our experiments. The simulator is cycle-accurate, allowing for accurate measuring. It can simulate different architectures with different memory models. We ran our experiments only on the 128-core Random Banked Memory architecture (rbm128), of which we used one *place* [7] of 64 cores. Each core is clocked at 1 GHz. due to simulation overhead, we could not run as many experiments as on the other platforms presented in this paper.

5.4.1 Implementation

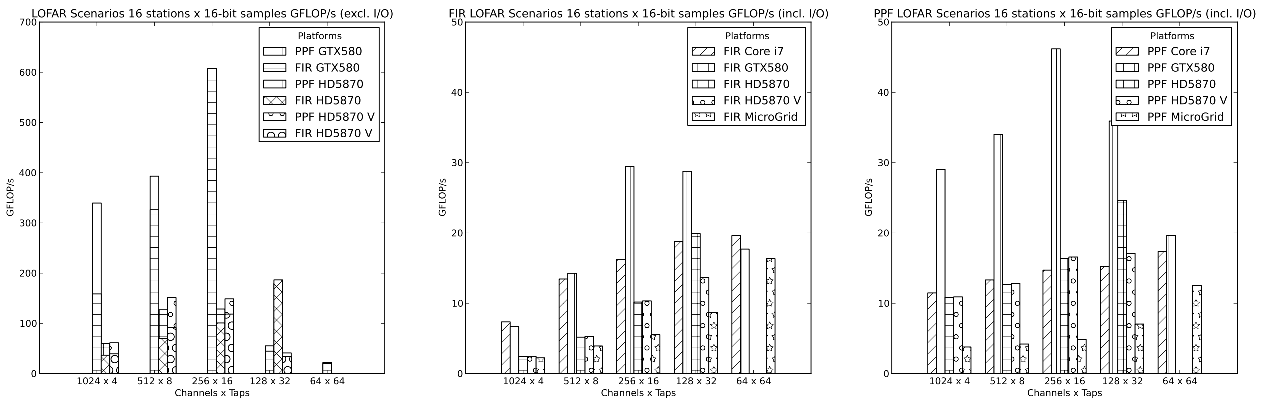
The implementation consists of two parts: the FFT and the FIR filter. We did not implement the FFT ourselves, but used the already available benchmarking implementation [8]. However, we modified it to use single precision floating point instead of double precision, and so it can run many FFTs in parallel, not just one.

The FIR filter reference implementation is an intentionally naive implementation, where each station, channel and tap has its own microthread. Ideally, this would be both the most efficient and easiest to program implementation, exploiting Microgrid's features as much as possible. The program creates a family of $N_{station}$ station threads which each run on a different core, each of which create a family $N_{channels}$ channel threads on the same core (to avoid the cache coherency protocol between cores), each of which in turn create a family of N_{taps} threads to compute the FIR outputs. Thus there are a total of $N_{stations} \times N_{channels} \times N_{taps}$ threads. The tap threads compute the output of both polarizations of the FIR filter at the same time (as in the CPU implementation), using shared parameters to sum the results. The station and channel threads do not need to communicate and only have global parameters.

The optimized implementation is similar, except that the tap threads are replaced by an unrolled loop inside the channel thread. This is very similar to our CPU implementation. Our experiments suggest that the Microgrid architecture is more efficient when using a high number of stations and taps, and a comparatively low number of channels. That means LOFAR scenario 1024 channels x 4 taps is the *worst case scenario*, and scenario 64 channels x 64 taps is the *best case scenario*. Microgrid benefits more from increasing the number of stations than the other platforms. This is the opposite of the GPU platforms.

5.4.2 Maximum performance

Unfortunately, we cannot calculate the maximum performance, because we do not know the memory bandwidth of the Random Banked Memory architecture. Moreover, Microgrid development has mostly switched to COMA (Cache-Only Memory Architecture), but we were unable to run our application on the COMA architecture due to bugs in the simulator. However, our results show that the FIR filter on Microgrid achieves 45 GFLOP/s in the best case (64 stations x 64 taps), which is 70% of the peak performance on



(a) (b) (c)

Figure 2: Performance of LOFAR scenarios: (a) GPUs excl. I/O, (b) FIR incl. I/O, (c) PPF incl. I/O.

Platform	Loop unrolling	Vector-ization	Batching	I/O page-locking
Core i7	++	+++	n.a.	n.a.
GTX580	+++	n.a.	+++	+++
HD5870	+++	+	++	+++
MicroGrid	++	n.a.	n.a.	n.a.

Table 6: Summary of impact of optimizations.

the configuration we have chosen (64 GFLOP/s). The full polyphase filter achieves 39% of the peak performance. Both are significantly higher than the other platforms we have investigated.

6. EXPERIMENTS AND RESULTS

In this section we compare the optimized implementations of FIR filter and the polyphase filter on the different platforms, using two criteria: performance of LOFAR scenarios and energy consumption.

LOFAR scenarios are the configuration of channel and taps used in practice by LOFAR. In these scenarios, when the number of channels doubles, the number of taps halves, and vice versa. This keeps the total FLOPs constant. The performance results are shown in Figure 2. Table 6 summarizes the impact of the optimizations we have applied.

To evaluate the energy consumption, we measured the energy consumption of the whole (desktop) computer using a Voltcraft Energy Check 3000. The results are presented in Table 5. We measured the minimum and maximum energy consumption of all LOFAR scenarios, but for readability we only show the average energy consumption of the 256x16 scenario. All measurements were taken with 16-bit samples. Finally, we show the amount of GFLOPs per Watt (GFLOPs/W) to gain insight into the actual energy efficiency. We have no measurements of the Microgrid architecture, as there is no hardware for it yet.

We observe that the CUDA implementation on the GTX580 gives the best performance in almost all cases. Note that the LOFAR scenarios do not achieve the highest possible performance. The highest performance we measured is 619 (FIR) or 576 (PPF) GFLOP/s with 64 stations x 1024 channels x 16 taps x 16-bit samples, excluding I/O transfers. Overall I/O has a huge impact on performance, reducing it by as much as 90%. The energy measurements show that the GTX580 is both the most energy efficient and power hungry device. Compared to the GTX480 it is not as energy efficient, but does achieve approximately 20% higher performance. Interestingly, in LOFAR scenarios where the occupancy is low (see Table 3), the power consumption is also low, because the device is underutilized.

The HD5870 does not achieve the performance expected from its hardware specifications. We expected the vectorized implementation to perform better, because it makes better use of the vector registers, but there is little difference. We believe this is because the ATI OpenCL compiler does not yet generate good enough code. Another reason might be that register spilling is more costly as the registers are 128 bits wide, compared to 32 bits on the GTX480/580. It consumes less power than the GTX480, but is only one third as energy efficient.

The Intel Core i7 is in a lower performance class than the GPUs, but can be used more flexibly because, unlike the GPU implementations, performance scales linearly with the number of taps, and there are fewer hardware limitations in general. It is the second most energy efficient platform.

The MicroGrid implementation excels in the specific case of 64 channels x 64 taps, which is precisely a scenario where GPUs are not efficient. In other cases it is not so efficient, but one should keep in mind that the MicroGrid architecture is still in research so the performance is expected to improve in later versions of the simulator, and eventually hardware.

Concluding, the CUDA platform for NVIDIA GPUs is at the moment the most promising many-core platform for the LOFAR polyphase filter. However, we have observed that the implementation is highly I/O bound. This is due to the low bandwidth (8 GB/s) of the PCI Express 2.0 bus. To make GPUs worthwhile to use, the I/O transfers latencies must be hidden by performing many operations per byte of input/output. This can be achieved by computing the entire LOFAR pipeline on the GPU, keeping the data inside the GPU in between pipeline stages.

7. CONCLUSIONS

We have discussed and compared the implementation of an efficient polyphase filter on the Core i7, GTX480/580, HD5870, and MicroGrid architectures. We have shown that

Platform	Idle (W)	256x16 I/O (W)	Min - Max (W)	GFLOPs/ W	256x16 No I/O (W)	Min - Max (W)	GFLOPs/ W
FIR Filter							
Core i7 920	84	154	150 - 154	0.10	n.a.	n.a.	n.a.
HD5870	110	186	184 - 188	0.04	219	219 - 229	0.46
HD5870 Vectorized	110	186	184 - 188	0.04	233	233 - 243	0.52
GTX580 CUDA	165	274	269 - 297	0.11	385	275 - 389	1.6
GTX480 CUDA	134	251	251 - 270	0.12	320	265 - 340	1.0
GTX480 OpenCL	134	220	220 - 240	0.13	292	247 - 304	0.96
Polyphase Filter							
Core i7 920	84	152	150 - 154	0.09	n.a.	n.a.	n.a.
HD5870	110	185	181 - 189	0.06	231	231 - 240	0.61
HD5870 Vectorized	110	185	181 - 189	0.06	234	234 - 243	0.64
GTX580 CUDA	165	280	276 - 295	0.16	420	299 - 425	1.20
GTX480 CUDA	134	256	252 - 270	0.19	345	282 - 362	0.99
GTX480 OpenCL	134	231	231 - 242	0.19	322	249 - 322	0.89

Table 5: Energy consumption on CPUs and GPUs. The left side shows the energy consumption with I/O transfers, and right shows without. Idle: Energy consumption while computer is idle. 256x16: Energy consumption of 256x16 scenario. Min - Max: Minimum and maximum measured energy consumption between all scenarios. GFLOPs/W: GFLOPs per Watt defines energy efficiency.

our novel implementation for the NVIDIA CUDA platform achieves very good performance and is most energy efficient of all investigated platforms. Moreover, our implementation is the first "real-world" application for the MicroGrid architecture.
Based on our results we conclude that CUDA-enabled GPUs is the best choice for the LOFAR polyphase filter, achieving the highest performance and the highest energy efficiency. As far as we are aware, this is the best performing polyphase filter implementation on CUDA-enabled GPUs so far.
In the near future, we plan to investigate alternative parallel FIR algorithms to achieve better performance for configurations in which our implementation is weak. Furthermore, more efforts should be put into implementing the whole LOFAR imaging pipeline on the GPUs, thus reducing the huge impact (up to 90%!) of the I/O transfers on performance. In the long term there are many research opportunities in integrating and testing the full LOFAR pipeline on GPUs.

8. REFERENCES

[1] CUDA Programming Guide. http://developer.nvidia.com.

[2] MicroGrid website. http://www.science.uva.nl/research/csa/microgrids.html.

[3] C. Carilli and S. Rawlings. Science with the Square Kilometer Array: Motivation, Key Science Projects, Standards and Assumptions. *New Astronomy Review*, 48:979–984, Sept. 2004.

[4] J. W. Cooley and J. W. Tukey. An algorithm for the machine calculation of complex Fourier series. *Mathematical Computing*, 19, 1965.

[5] M. Frigo and S. G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384 vol.3. IEEE, May 1998.

[6] B. K. Hamilton. Implementation and Performance Evaluation of Polyphase Filter Banks on the Cell Broadband Engine Architecture. Master's thesis, University of Cape Town, October 2007.

[7] C. Jesshope, M. Lankamp, K. Bousias, and L. Guang. Implementation and evaluation of a microthread architecture. *Journal of Systems Architecture*, 55:149–161, 2009.

[8] C. Jesshope, M. Lankamp, and L. Zhang. The implementation of an SVP many core processor and the evaluation of its Memory Architecture. *ACM SIGARCH Computer Architecture News*, 37, No. 2, May 2009.

[9] D. Miles. Compute intensity and the FFT. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, Cray Res. Superservers, Inc., Beaverton, OR, USA, November 1993. ACM.

[10] C. Neau, K. Muhammad, and K. Roy. Low complexity FIR filters using factorization of perturbed coefficients. In *Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001. Proceedings*, pages 268–272. IEEE, 2001.

[11] J. Pettersson and I. Wainwright. Radar Signal Processing with Graphics Processors (GPUs). Master's thesis, Uppsala University, January 2010.

[12] J. G. Proakis and D. G. Manolakis. *Digital Signal Processing*. Pearson Prentice Hall, fourth edition, 2007.

[13] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo. SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93(2):232– 275, 2005.

[14] A. Smirnov and T. cker Chiueh. An Implementation of a FIR Filter on a GPU. *ECLS*, 2005.

[15] R. V. van Nieuwpoort and J. W. Romein. Correlating Radio Astronomy Signals with Many-Core Hardware. *Accepted for publication in Springer International Journal of Parallel Programming*, 2009. Special Issue on NY-2009 International Conference on Supercomputing.

[16] S. Williams, A. Waterman, and D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Communications of the ACM*, 52, No. 4, April 2009.