ADAPTING A SDR ENVIRONMENT TO GPU ARCHITECTURES

Pierre-Henri Horrein (CEA, Leti, Minatec, Grenoble, France; pierre-henri.horrein@cea.fr);
Christine Hennebert (CEA, Leti, Minatec, Grenoble, France; christine.hennebert@cea.fr);
Frédéric Pétrot (TIMA, Grenoble-INP, Grenoble, France; Frederic.Petrot@imag.fr)

Abstract

This article studies different solutions to use the Graphics Processing Units computing power for the Software Defined Radio environment GNURadio. Two main solutions are considered. In a first implementation, which has already been studied, the GPU is only considered as a fast additional processor, with specific algorithms implemented. In a second implementation, the specific characteristics of the GPU are taken into account, and the environment runtime is redesigned to accommodate for the presence of the GPU. Instead of using GPU optimized algorithms, this second solution uses classical algorithms on multiple arrays.

Both solutions are implemented and compared on different operation types, and on a complete operation sequence. It is clearly shown that using the second solution can provide performance improvement, while the first one is inefficient for SDR applications.

Keywords: GPGPU; SDR; GNURadio

1 INTRODUCTION

The fast development of wireless networks in the last years has modified the implementation choices. While network devices used to process the signal were traditionnally implemented as hardware, hard-wired components, the aim now is to have more and more flexible implementations, in order to follow the improvements. A possible solution to reach this flexibility is Software Defined Radio (SDR).

A SDR terminal takes the border between software and hardware as low as possible. While traditional devices implement all computationally intensive tasks in hardware, thus pushing the border somewhere in the protocol layers, SDR devices implement as few operations as possible in hardware, quickly sampling the signal and using processors to process this signal. As can be expected, the main limitation for this promising implementation is the required processing power. Signal processing operations are usually demanding operations. While optimizations could be done by designing dedicated SDR applications, this would limit flexibility. SDR environments are thus the preferred option when designing SDR application.

To cope with the required processing power, General-Purpose computation on Graphics Processing Units (GPGPU) is a possible solution. It allows processing of data at a very high rate, providing a very high computing power, well suited to image processing algorithms. This is achieved by using Graphics Processing Units (GPU) as conventional processing units. GPGPU can increase the possible throughput, and it can ease the task of associated General Purpose Processors (GPP), freeing them for other applications or for protocols.

But benefiting from GPU is not a straightforward operation. SDR environments are usually designed for CPU architectures. CPU platforms are task parallel platforms, meaning that several tasks can be run simultaneously. GPU architectures are Single Instruction Multiple Data (SIMD) architectures, designed to run simultaneously a single instruction on a large set of data. This mismatch in the computing model used, as well as the required framework for GPGPU which leads to high memory transfer overhead and centralized management, means that deploying an SDR environment on a GPU platform may not improve performance. The aim of the work presented here is to define and compare different possible software architectures for an efficient execution of a specific SDR environment, GNURadio [1] on the GPU, in order to keep the flexibility and ease of use of the environment, while benefiting from the computing power of the GPU.

The paper is organized as follows. In Section 2, a presentation of the GPU architecture is given, as well as the SDR environment used, the GNURadio environment, and some examples of previous works. In Section 3, the different solutions which were considered are presented. Results for the standard implementation and for both solutions are presented in Section 4 and conclusions are finally drawn in Section 5.

2 ENVIRONMENT AND PREVIOUS WORK

2.1 The OpenCL environment

The GPU environment used in this work, Open Computing Language (OpenCL)[2] is a unified computing environment standard, published by the Khronos Group, and designed to enable heterogeneous parallel computing.

OpenCL is based on a hierarchical representation of the computing platform, as presented in Figure 1. The platform is divided between a host, usually a classical Central Processing Unit (CPU) and Computing Devices. These computing

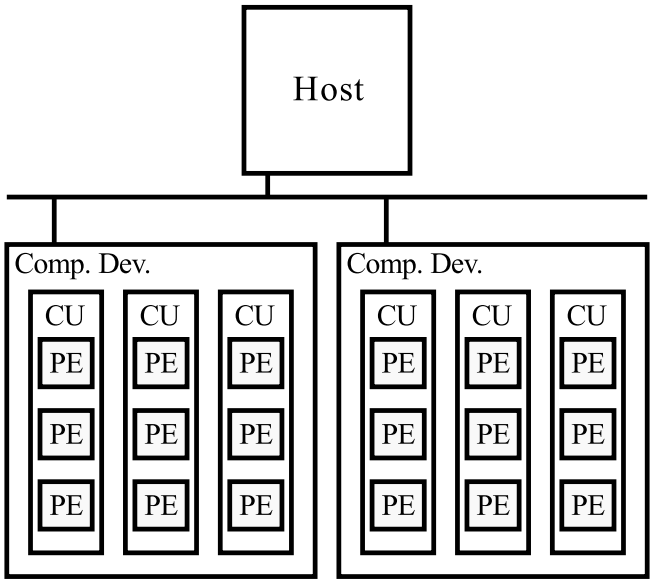

Figure 1. OpenCL platform representation

devices are made of Computing Units (CU), and each of these units is built from Processing Elements (PE). The PEs are the basic block of an OpenCL platform. Computing Units are based on the Single Instruction Multiple Data (SIMD) model, with all the processing elements executing the same stream of instructions on a data vector. The OpenCL standard also allows the Single Program Multiple Data model (SPMD), in which all the PEs execute the same program, but are not linked to each others.

Processing in OpenCL is divided in two parts:

- the kernels, which are executed on the Compute Devices,
- the host program which is executed on the host, and which controls the execution of kernels on Compute Devices.

The data set on which a kernel must be executed is divided using an index space. A kernel is instantiated for each of the single element. This instantiation creates the work-items. Each work-item of an index space executes the same program. Work-items are then organized in work-groups. A work-item thus has a global identification number (ID), in the complete index space, and a local ID in the work-group. Work-groups are submitted for execution to the Compute Units. Work-items of a work-group are executed on the PE of the associated Compute Unit. All commands to the GPU, such as memory transfer requests or kernel executions, are issued using a command queue.

Possible programming model for OpenCL programs are Data Parallel or Task Parallel. Data Parallel is the preferred model, since OpenCL architecture is designed as a SIMD processing platform. Task Parallelism is obtained using a single work-item, with a parallelization over the different CUs.

2.2 GNURadio architecture

The aim of this work is to use OpenCL and its GPGPU capabilities for a SDR environment. The environment chosen here is GNURadio. GNURadio is a project offering a complete framework for Software Defined Radio and Signal Processing applications. A SDR application can be represented as a sequence of processing blocks, with communication channels such as FIFOs between each block. These flowgraphs, as they are called in GNURadio, can be built using predefined blocks.

GNURadio provides several elements:

- basic processing blocks, with a hierarchical classification, and templates to easily develop new blocks. These blocks are developed in C++,
- a runtime part, comprising a scheduler, communication channels between the blocks, or a flowgraph interpretor,
- input/output (I/O) libraries, offering interfaces with possible sources and sinks of the platform, such as the sound card, or the Universal Software Radio Peripheral (USRP) and its extensions.

Two possible scheduling methods are actually implemented in GNURadio:

- the first method uses a single process, and switches from block to block,
- the second method uses one process (thread) per block, with blocking communication channels.

The main advantage of environments such as GNURadio lies in the runtime part, and in the flowgraph interpretor. It also offers a wide variety of existing blocks.

2.3 Previous work

The use of GPU computing for SDR application is not a new field of research. SDR applications are demanding applications, with high computing power requirements, and the increase of the offered throughput in wireless standards means that classical CPU approaches are not able to stand the data rate anymore.

A first example of an existing solution is given in [4], in which a hardware platform and software architecture are given in order to enable GPU computation of SDR applications. This study is interesting, as it proposes a complete WiMAX modem implementation, but the software platform is not GNURadio, and the main aim is to present the global architecture, not details on GPU management.

On the implementation side, [3] and [5] propose two complete studies of implementation of SDR on GPU. In [3], polyphase channelization outside of any unified environment is studied. Significant improvement can be seen in the results, but the targetted application is well suited to GPU computation. In [5], Standard Communication Architecture is studied, and possible integration of GPP elements in an architecture. Results are very interesting on large set of samples. The proposed solution of [5] is similar to the Direct Mapping solution in this study.

The next sections present two approaches to the proposed problem, and the results of these approaches when compared to a classical CPU implementation.

3 SOFTWARE RADIO USING OPENCL

3.1 Design of a GNURadio application

A GNURadio application is a set of processing blocks, linked together using FIFO channels, and can be represented as in Figure 2. Each block has a processing function, designed to process arrays of values, and to produce values. An example of a processing block is a N-points Fast Fourier Transform (FFT). The processing function of the block takes as inputs N-values arrays, and outputs N-values arrays representing the FFT of the inputs.

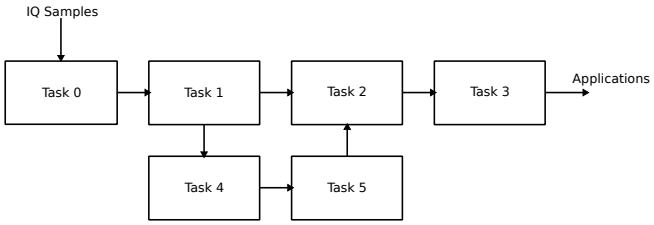

Figure 2. Example application representation for GNURadio

The FIFO channels are inter-blocks communication elements. They are designed to be used as links between processing blocks, and are managed by the runtime part of GNURadio. This runtime is used to run the processing functions of the blocks. When the block is initialized, the estimated number of input values required to produce a given number of output values is processed. The runtime monitors the filling of buffers in order to call the processing function when required.

3.2 Direct Mapping Solution

The aim of this study is to make use of the GPU power in the GNURadio environment. The first straightforward solution is the direct application mapping solution. This is the strategy adopted in [5] for example, and in most previous studies.

When using the Direct Mapping solution, the execution environment is exactly the same as in a CPU-only execution, except that some of the blocks are processed by the GPU. GNURadio is clearly divided in a runtime part and a processing part. Each of the processing block is thus made of integration in the runtime environment, as well as a process function used to actually process data.

In a first implementation, the runtime is kept, but the process function of the block is modified in order to use the GPU for processing of data. This means that the GPU is only used to speed up the actual computation, while all the control is done on a CPU, which thus becomes the OpenCL host. The aim of this implementation is to benefit from the GPU power to speed up atomic operation: specific algorithms are implemented, which make use of the many-cores architecture of the GPU to have an efficient basic operation. This is represented in Figure 3.

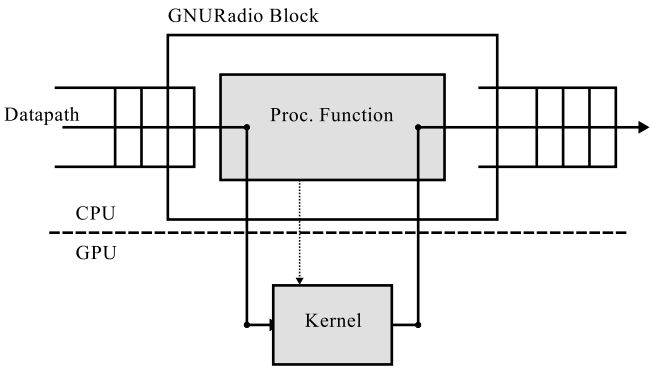

Figure 3. Direct Application Mapping Solution

The main advantage to this solution is its simplicity. Nothing needs to be changed in the environment. This solution could basically be implemented in any SDR environment. The only additional feature required is OpenCL initialization. It also easily allows hybrid GPU/CPU radio applications, since communication between the processing blocks is done in the CPU subsystem, and data is only given to GPU when needed (there is always a copy in system memory). This opens the door to the integration of GPU as an additional, very efficient, processor, but still allows CPU operation when GPU implementation is inefficient.

This solution induces a very high memory transfer overhead, since data to be processed must be transfered from the CPU to the GPU, and the results must be written back. If the results must be used as inputs to another GPU block, a new memory transfer must be done. The cost of a memory transfer in a GPU architecture is such that it might be better to process even unefficient algorithms on the GPU, in order to avoid useless memory transfers.

This means that in order to take into account the presence

of the GPU into the environment, and to avoid useless memory transfers, the runtime must be modified. In this modified runtime, communication channels between the blocks, which were untouched in the straightforward implementation, are modified to accomodate for the GPU presence. Four types of channels are implemented:

- CPU to GPU channels, when source block is run by the CPU and sink block is run by the GPU,
- GPU to GPU channels, for communication between GPU blocks,
- CPU to CPU channels, for communication between CPU blocks,
- and GPU to CPU channels, for communication between a GPU source block and a CPU source block.

The runtime is subsequently modified, in order to take these new channels into account. Processing blocks are still developed using the separation between runtime and processing. The difference between both implementation lies in *process* function implementation. Instead of sending required data to the GPU, process, and copy the results, the function is only in charge of activating the kernel, as can be seen in Figure 4.

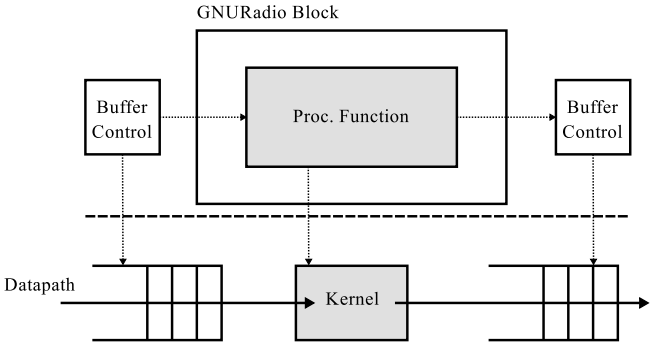

Figure 4. Direct Application Mapping Solution with special buffers

Channels are implemented as mixed CPU/GPU entities. For CPU to CPU channels, the classical GNURadio buffer implentation is kept. For GPU to GPU channels, index management is done in the CPU, while data is kept in the GPU memory. Transfers between CPU and GPU are done explicitly, using a special transfer block. This block does nothing except the data transfer.

The main advantage of this second implementation is that no more memory copies are required. Transfer can be done asynchronously to the GPU memory, and two consecutive blocks in GPU can keep memory in the GPU instead of transfering data twice. But using this solution, mixed execution of a single block by a CPU and a GPU is not possible anymore, and the target of the block must be specified when instantiating the application.

For both solutions, another important point of this implementation is that with OpenCL, the bigger the dataset, the more efficient the GPU becomes, as can be seen in Section 4. Taking the example of a FFT, it becomes truly efficient when the FFT is run on 32768 points, which is not common in radio applications.

3.3 Making GNURadio data parallel

The Direct Mapping solution, while attractive with its simplicity, suffers from drawbacks.

- In order to be efficient, a GPU application must maximize the use of the GPU. For the previous solution, this means working on large sets of data, which are uncommon in radio applications.
- While some SIMD processing blocks can benefit from a GPU implementation, other data-dependant functions are completely unefficient when executed by a SIMD processor.

In order to provide a solution to these two points, another software architecture has been designed, which tries to transform any GNURadio application into a data parallel application working on large sets of data. The main idea of this solution is that, instead of keeping the input by input model, with OpenCL kernels designed in order to locally optimize a function, a large number of data are processed simultaneously, by kernels implementing the complete function, as represented in Figure 5

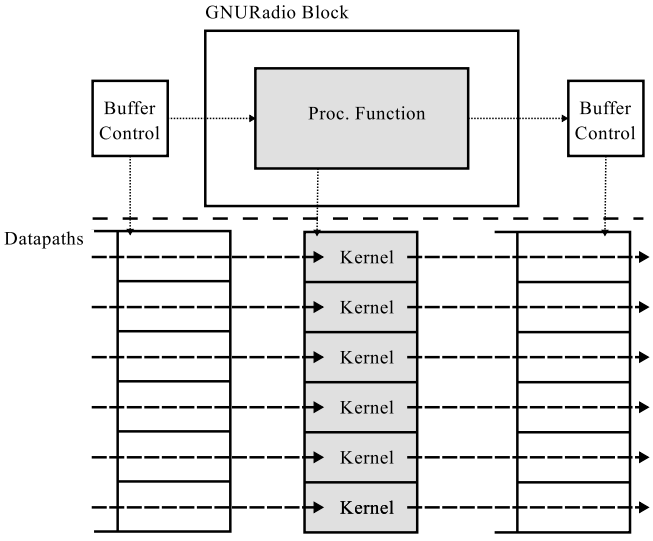

Figure 5. Data parallel solution

This solution implies modification of the runtime and of the processing blocks. The runtime must be adapted to the accumulation of data in the inputs, by allocating bigger buffers for example, or by waiting for more data before processing. It also implies that the latency may increase. But it has several advantages.

- It is not necessary to find a GPU-optimized version of all operations in order to be efficient, since the kernels implement the same algorithms as on a CPU.
- GPU usage is maximized.
- Even data dependent operations can be processed and improved, as long as it is not a stream operation. This means that, as long as the data-dependency characteristic is only true for specific known vectors, it is possible to run them concurrently. On the contrary, if the data-dependency is a stream characteristic, meaning that all the operations will always depend on previous results, with no reset, then it is not possible anymore.

But memory usage can quickly become a problem, since the need to store more input may lead to unsufficient memory space. And depending on the requirements, the latency induced by this solution may be too high.

4 RESULTS

4.1 Presentation

In order to validate the two approaches, four applications have been developed and executed:

- a Fast Fourier Transform (FFT), for which efficient SIMD algorithms exist,
- an IIR, for which no SIMD algorithms exist, due to a feedback value in the processing,
- a demapper block, to see the effect of GPU on data manipulation blocks,
- a sequence of all three blocks (mapper, IIR, FFT, iFFT, IIR, demapper) in order to check the efficiency of the solution in a "real" environment.

For each of the applications, results are given for CPU approach, and for the pertinent GPU approaches. The performance indicator used here is the time required to process 100,000 arrays.

The computer used for the experiments is based on an Intel Core i5 760, which is a quad core CPU at 2.80 GHz, with 8 MB of cache. The GPU is a NVidia GTS450, which is a cheap GPU, with 128 cores (processing elements), organized in four 32 cores multiprocessors (compute units). Core frequency is 738MHz. Available memory for the GPU is 1GB.

The implementation is not yet fully operational. GPU buffers are not fully implemented in the results presented. While this is not an issue for single operation experiments, it has a big impact on the sequence, since for each block, in order to have correct results, data must be transferred from the CPU to the GPU and back.

4.2 FFT

The first operation implemented is the FFT. This operation is one of the most used operation in radio applications. Results are presented in Figure 6 for small, usual array sizes. Results show the time in milliseconds required to process 100,000 2^N FFT, with N varying from 5 (32 points) to 13 (8192 points).

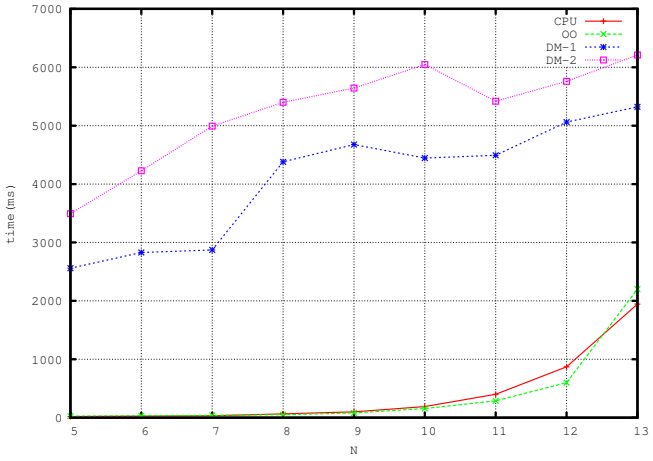

Figure 6. Results for FFT

As can be seen on the results graph, the direct mapping solution is not an efficient solution. Even more surprising, the version with dedicated buffers between the blocks (DM-2) is less efficient than the version with explict transfers in the blocks (DM-1). This is due to the overhead of buffer management. Since the FFT application uses only one block, the amount of memory transfer stays the same. The second solution is useful only when multiple blocks are used. The DM solution becomes more interesting than the CPU for a FFT size of 32768 points, which is unrealistic in a radio application.

On the contrary, the data parallel solution (OO for overall optimization) is more efficient than the CPU solution for FFT less than 4096 points. For a classical 1024 points FFT, the CPU solution takes 190 ms to process the 10^5 FFT, while the GPU solution takes 156 ms to process the same FFTs, which is a 18% gain.

4.3 IIR

The second application is an Infinite Impulse Response, which is a data dependant operation, meaning that the result

for element i of an array depends on the result for element $i-1$. Results are not drawn for the Direct Mapping application, which yields very bad results, since no optimized algorithm exist for the IIR, and all the IIR computation is done on a single processing element.

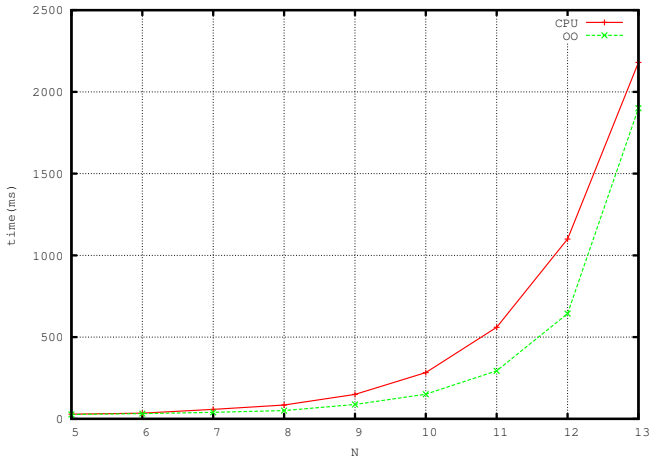

Figure 7. Results for IIR

The IIR implemented is a fixed size IIR, working on arrays of 2^N samples, and no dependency between two arrays. Once again, results presented in Figure 7 are time results for 100,000 arrays. The GPU solution is also better than the CPU solution. GPU time for an array of 1024 samples is 151 ms, while the CPU time is 283 ms, which gives us a 47% gain in time using the GPU. The decrease of the gain when the size of the array increases is due to the limited buffer size. When the size increases, less arrays can be stored in the buffers between the blocks, which means that the GPU is not fully used. As an example, the NVidia profiling tool for the GPU shows a 97% GPU use for a 1024 samples array, while this usage falls down to 51% for a 8192 samples array. If sufficient memory is available, and big arrays must be processed, increasing the buffer size increases the gain. Buffer size must be chosen to be able to store 128 arrays in order to maximize GPU usage.

4.4 Mapping/Demapping

The last single operation is a demapping operation for a QPSK modulation. The aim of this operation is to map the received samples with the associated binary signification. In a QPSK modulation, a sample is used to represent 2 bits. The demapping operation is not a signal processing operation, it only involves data manipulation.

Results for demapping of 100,000 arrays of size 2^N are presented in Figure 8. Results for the Direct Mapping solution are once again disappointing for array sizes below 32768, and are thus not shown on the Figure. As an example, demapping 100,000 1024 samples arrays using the DM-1 solution takes around 3 seconds, which is 100 times the time required by the OO solution.

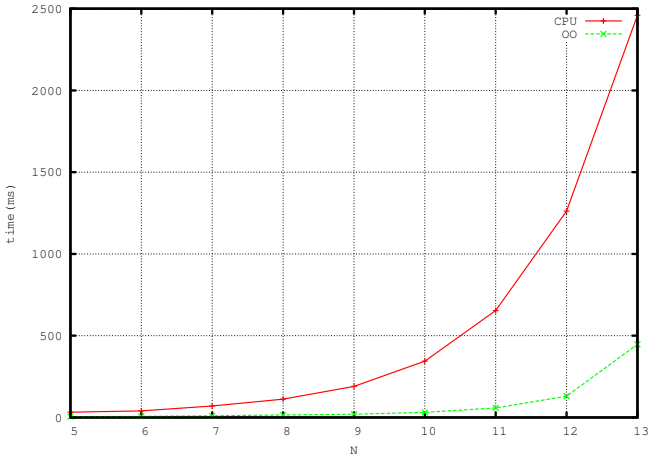

Figure 8. Results for demapper

The GPU is once again more efficient than the CPU, with gains going up to 90% for the 1024 samples array (344 ms for the CPU, versus 31 ms for GPU implementations). The efficiency of the GPU for the demapping operation is clearly shown in these results.

4.5 Sequence

Finally, in order to evaluate the performance of the GPU in the case of a complete application, all three previous operations are performed on the arrays. Results for the Direct Mapping solution are not shown, since the efficiency of both solutions is very bad when compared to the CPU solution (almost twice the time for the DM-1 solution, and 1.5 times the required time for the DM-2 solution).

Results for the OO solution and for the CPU are presented in Figure 9.

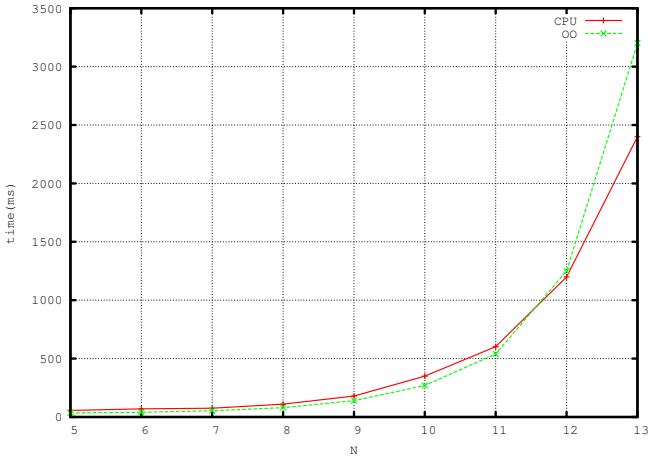

Figure 9. Results for the sequence of operations

The CPU solution benefits here from the task parallelism. It can be seen that results for the complete sequence are similar than results for the demapper. The GPU does not allow task parallelism, thus leading to cumulated processing times. It is also interesting to notice that timing results include reading test samples from file and writing results to file. Since the GPU buffer is not fully implemented yet, processing time includes useless data transfers.

Results still show that the GPU solution is more efficient than the CPU solution when the array size is lower than 4096 points. When this size is reached, the CPU becomes more efficient. Gain for 1024 points is 22% (350 ms versus 270 ms).

5 CONCLUSION

This study has shown two possible methods to include GPU in the GNURadio environment. The direct mapping method, which uses the GPU as an efficient CPU, with no real integration in the environment, is unefficient, and yields worse results than the CPU version. This was the method discussed in the GNURadio mailing lists.

The new method proposed in this article does not use GPU specific optimization for the block implementation. Instead, it focuses on processing several arrays at the same time. Since radio applications are mainly stream applications, with the same sequence of operations applied to all arriving samples, it is possible to extract data parallelism from the application. The resulting SDR environment performs around 20% better than the CPU application, using a powerful CPU and a cheap GPU. These results are expected to become much better when a fully functional GPU buffer is implemented in the environment.

Several improvements and discussions are considered based on this study. First of all, a working GPU buffer implementation is currently under way. Given the early results, it is expected that processing time for the complete sequence application can be divided by at least 2 for big arrays. Once the buffer implementation is complete, the focus of this study will be integration in an embedded environment, with two main questions:

- what will the results be in an embedded environment ?
- given the constraints of embedded environments, especially in terms of power consumption, is the GPU a viable solution ?

REFERENCES

[1] GNU Radio. `http://gnuradio.org`.

[2] The OpenCL Specification, Version 1.1, September 2010.

[3] G. Harrison, A. Sloan, W. Myrick, J. Hecker, and D. Eastin. Polyphase Channelization utilizing General-Purpose computing on a GPU. In *SDR 2008 Technical Conference and Product Exposition*, 2008.

[4] J. Kim, S. Hyeon, and S. Choi. Implementation of an SDR System Using Graphics Processing Unit. *IEEE Communication Magazine*, pages 156–162, March 2010.

[5] J. G. Millage. GPU integration into a Software Defined Radio Framework, 2010.