INAF-Osservatorio Astrofisico di Arcetri

The software spectrometer `SpectralGpu`

E. Giani, C. Baffa, G. Comoretto

Arcetri Technical Report N° **8-2013**

Abstract

The SpectralGpu *project is aimed at the development of a software spectrometer for the radio wawelengths range using the computation power of the recently available GPU video boards.*

Figure 1: *Logo del progetto* SpectralGpu

1 Forewords

Historically the design of a Radio wavelength Spectrometer ha assumed many different forms, for example the Acousto-Optical Spectrometer or the filter bank arrays, but now it is dominated by digital electronic correlators.

In very recent times, the easy and cheap availability of graphical parallel processor (GPU) for the consumer market, has expanded once again the range of possible choices. The computational speed of one recent GPU board can outperform by orders of magnitudes the maximum speed of the original Cray-1 and Cray-2 supercomputer (around 500-1500 GFlops versus 250 MFlops for Cray-1 and 1.9 GFlops for Cray-2[2, 3]).

With those fast boards we can conceive a full software spectrometer, `SpectralGpu`, and the present report describes its development.

2 Functional Design

The `SpectralGpu` spectrometer gets its input from the digital receiver by means of a dedicated 10 Gbit Ethernet line. The receiver output data rate is tunable from 0.5 to 255 MS/s (10^6 samples/second) and the native format for each channel is a complex 8+8 bit signed integer (real ad imaginary parts as `signed chars`).

The desired spectral resolution ranges from 1 to 10 m/s. At a wavelength of 1 cm this translates to about 1 KHz. With a design band of 100MHz, this requirement implies a 64K Fourier Transform.

To limit the spectral leakage a windowing[1] must be applied before the Fourier Transform. The *windowing function* can be chosen to enhance different characteristics of the spectra.

The integration can be accomplished by means of a simple accumulation of the moduli of the different channels of the transform output data.

[1] multiply of data with a special *windowing function*

3 Implementation details

The data source is connected to `SpectralGpu` by means of a fast 10GHz Ethernet link and we use a `dedicated driver`[1] to feed data to the GPU engine. The network board is a Myricom Myri 10G-PCIE-8B-S+E, using a PCI Express X8 bus, with an expected throughput of 300MSamples/s (600 MB/s bandwidth). The actual maximum bandwidth, due to memory bottleneck, is around 400MHz. Full speed can be achieved by using a custom driver.

During the design of the `SpectralGpu` project we choose to use coprocessor GPU boards which have many simple elementary CPUs, as those manufactured by `ATI`. We evaluated that our problem, involving many simple computations, is best suited to simpler and more numerous engines.

That choice implies the use of OpenCL programming environment, with a much steeper learning curve in comparison with `NVIDIA`'s CUDA. At the end, however, we got a finer grain control over the coprocessor operations. After a somehow longer start-up phase, the development proceeded fast and the GPU portion exceeded the planned speed already during the firsts development iteration cycles, marking a clear success in this milestone.

We have successfully tested our code on a NVIDIA 8300, using the OpenCL portion of CUDA SDK. As expected this combination has a somehow inferior performance, compared with the ATI boards, probably dues to a less efficient driver.

4 Hardware and software platform implementation

The heterogeneous platform used to develop this work includes an Intel Core i5 processor and a graphics card with an ATI Radeon HD 7700 (Capeverde). The graphics card is connected to the mother board using a 16-lane PCI-express interface.

In a second step, we have installed a second graphics card: an ATI Radeon HD 6700 (Juniper) using a 4-lane PCI-express interface.

The OpenCL implementation packaged with the AMD-APP-SDK-v2.8-ln32 is used for the CPU and the GPU. The device driver to handle GPU is the Catalyst version 12.4, with the fglrx ATI proprietary Linux Driver module version 8.96.4. The OS running on the host is Linux Mint Maya.

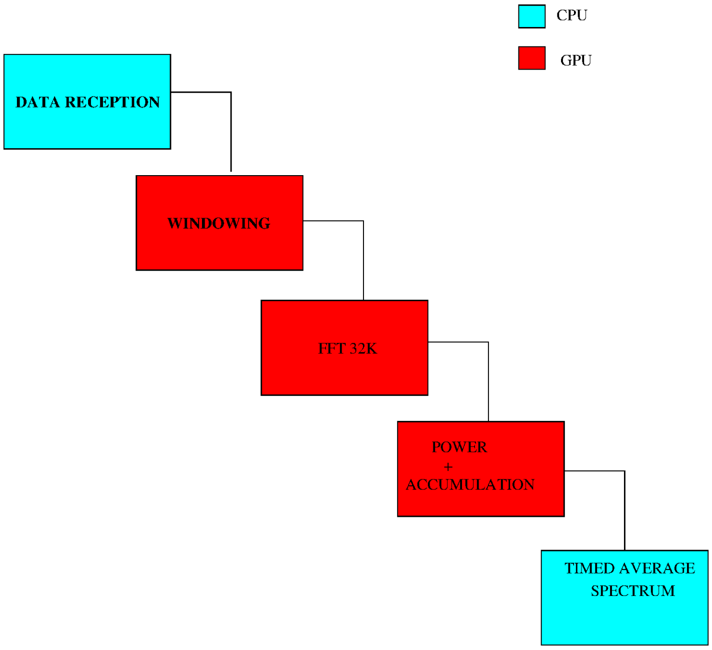

Figure 2: *CPU-GPU workload distribution for* `SpectralGpu`

The target PC will host a AMD processor and an ATI Radeon HD 7970 (Tahiti), with the same software configuration.

5 The OpenCL `SpectralGpu` program

A generic OpenCL program is divided in two parts: one executed on the "parallel intensive" *device*, generally a GPU, and other running on the host, generally the CPU

The application running on the CPU, named also *host application*, is a standard C or C++ program where the communications between the host and the graphics device(s) are realized using the commands implemented by the OpenCL API.

The code executed on the *device*, instead, relies on some special functions called *kernels* which are coded in the OpenCL Programming Language.

The figure 2 shows how the `SpectralGpu` program tasks are distributed across the two processing architectures of the platform: the CPU and the GPU.

The host program collects data from the fast ethernet link and transfers it to the GPU. Then the GPU, using three separate kernels, computes the FFT and the power spectrum.

Once this triple task is completed, the results are transferred back to the CPU where the time-averaged spectrum is computed. The CPU part does not use an OpenCL kernel but a regular C code to perform the reduction step.

Data is received by the program in the form of couples of 8 bit signed integers (real and imaginary part, see figure 3) and are written into the global memory of the GPU in blocks of 32KSamples.

On every iteration a fixed number of blocks are loaded and processed in parallel, and this number is chosen to maximize the global throughput. The optimal number depends on GPU architectures: we got the maximum computational speed processing from 128 to 512 blocks at time.

5.1 The adopted working strategy

Our preliminary tests have proved that the application is limited by memory transfer speed, so an important point of overall program efficiency is the optimization of the transfer data rate from CPU to GPU.

We experminted the use different tecniques for **device** memory access optimization, with marginal gains. A more fruitful solution would be overlapping the GPU kernels computations with other more time-expensive operations, as data reading from link and host-to-device memory transfers.

Using the perfomance tools supplied with the AMD-APP-PROFILE package, we have seen that the GPU has a low time occupancy of its compute units (CU), if compared with the time spent in fetching data. This means that for part of the working time the GPU CUs are idle, and we need to explore a more efficient approach.

Taking into account the previous considerations we have developed a multi-threading host program. It relies on POSIX threads setup to get a framework where multiple light processes are executed on the CPU cores. Also, to take advantage of the GPU computation capabilities, multiple instances of the same kernels, each with multiple work-items, are enqueued for parallel execution in on the same GPU or on two separate ones.

5.2 OpenCL initialization

The host application starts with the initialization of the OpenCL framework, checking the OpenCL capable platforms presence, the type of *devices* to use and the number of them. Next, to gain a deep control of the graphic hardware parameters, the program initializes the AMD Display Library (ADL), an interface designed to access low level information related to ATI graphics cards associated with the system.

To manage an OpenCL capable device, the host program has to create at first an OpenCL context, an object defining the entire OpenCL enviroment, and then a command queue to issue the OpenCL commands to the device selected under the context.

The `SpectralGpu` program can start up to two threads for available gpu, to fulfill the requests coming from the configuration options passed as argument to the application.

Each thread creates its own command queue, so each queue is receiving and processing commands in parallel with other active queues.

If the system includes more than one GPU, a thread is started for each device. Each device has its own command queue and the application achieves more hardware level parallelism.

The command queues can share a single context or each queue can be associated to a different one. The first solution is faster and requires less memory, but the access to a single context across multiple threads may generate problems.

To be on the safe side, the `SpectralGpu` application creates an OpenCL context for each thread running on the active GPU. Each one is configured to run more than one thread per GPU, and it uses multiple devices for computations.

The kernel functions executed on the GPU devices are stored in a OpenCL source file whose code is loaded and compiled at run-time to generate the device specific executable.

A central part of the application is the data transfer to and from the device. The host program allocates a set of memory buffers on the gpu device. These are associated with the context device, not with the devices itself, so each thread has its own set of memory objects to which the host program has to interact for I/O.

This implies the application needs to implement some synchronization mechanism to coordinate the access to data stored in independent buffers.

5.2.1 The command queue type

The OpenCL standard supports two different type of command queues: the *in-order* type and the *out-of-order* one.
The first is a simple implementation of a sequential execution queue. The commands issued in a in-order type queue are sequentially executed and the execution of a command on the device starts at the completion of the previous one.
On the other hand, using out-order command queue type, it's possible to parallelize execution of commands. In out-of-order execution mode there is no guarantee that the enqueued commands will finish execution in the order they were queued: an order of execution can be defined by the programmer using events.
It has to be noted that the OpenCL implementation which comes with the current AMD platform supports *only* the in-order type command queue, as can be seen running the command *clinfo*.

6 Queuing, global synchronization and memory consistency in OpenCL

Under the OpenCL framework, the host is the central point for all application control logic.
The OpenCL commands are enqueued asynchronously onto a command queue and the associated operations are executed at some point in the future. The host program has no control on this: from the point of view of the host program the compleation of a command is guaranteed only at synchronization points. These are:

- the clFinish command that blocks until an entire queue complete execution
- waiting on completion of a specific event
- execution of a blocking memory operation

On the OpenCL model, any memory object that can be shared between multiple enqueued commands is guaranteed to be consistent only at synchronization points. For commands belonging to the same command queue, the consistency is guaranteed for an in-order queue type or the other case, through the communication of an event from the command generating it and the one waiting on it in out-of-order case. But this internal consistency is not visible to the host: to get memory consistency the host program has to use one of the previous blocking mechanism.

The understanding of the synchronization mechanisms in OpenCL is very important, especially when working with multiple commands queues, as in the case of the `SpectralGpu` application. It is important to note that synchronization using OpenCL events can only be done for commands within the same context, which is not our case because the `SpectralGpu` program generates separate context for each thread, and hence for each device.

But the `SpectralGpu` application behaviour is simpler because data is shared only between the CPU and the GPU threads, not between the different OpenCL contexts (see§ 5.2).

7 The host program structure

The `SpectralGpu` application starts a set of POSIX threads.

The default action is to start three threads: one to collect the data from the ethernet link, the second to control the GPU temperature and the last to handle the communication between the host and the GPU device.

If the working platform has more than one discrete gpu active, the program starts one more thread for each extra graphics device.

We plan to add one more thread whose main task will be the output data collection and reduction. The interim solution is to distribute this activity to each single `gpu thread`, using a global shared buffer controlled by a mutex.

7.1 The reader thread

This task get the receiver data from the fast Ethernet link. Data are stored into a memory buffer organized as four elements FIFO (four chunks).[2]

[2] This value can be changed through the argument options *-s*. The default value 4 represents a good compromise.

The size in bytes of each chunk is equal to the overall length of the block of data sent as input to the FFT routine. Two semaphores control the availability of the resources for reading or writing operations, while a mutex governs the access to the FIFO itself.

As stated before, the program will receive data from the Ethernet link, but, during the development phase, the acquisition is simulated by means of a file stored on the hard disk.

The I/O speed of the hard disk is lower than that of the fast Ethernet board, so the computation of the power spectrum of each segment of data is faster than the reading operation from file. As a consequence, the `gpu thread` has to wait frequently to access to new data.

We do not get benefits by working with two threads running on the same GPU. Even when the threads overlap, we get the same execution time with one or two `gpu threads`. We evaluate this results come from the resources exhaustion of our gpu board.

7.2 The temperature monitoring thread

This task periodically reads the GPU temperature and, in case, operates a tuning of the fan parameters to maintain the temperature stable.

This process uses the AMD Display Library (ADL) delivered as part of the Catalyst display Driver package.

The APIs library make also available handles to modify the settings of the different performance profile levels. It is, for example, possible to specify the memory clock or the main processor bus clock.

If the platform addressed by the OpenCL program has more than one GPU active and operating, one temperature thread is created for each of these devices.

7.3 The GPU thread

The `gpu thread` enqueues both the OpenCL memory commands and the kernels into the corresponding commands queue.

Our system works as an iterative process: the same set of kernels is scheduled to run against each successive sets of input, thus producing a series of consecutive output data.

During each iteration, the new input data, collected by the `reader thread`, are transferred to the device memory by the *clEnqueueWriteBuffer* command against the associated memory object, allocated during initialization phase. Only when the write command is completed, the memory pointed by the host buffer can be reused by the host application.

Instead of using blocking commands, the program relies on OpenCL event objects, used to query the execution status of the commands.

An event object is created for each read or write command issued into the command queue. These event objects allow the host program to get information about the status of the command, for example its completion, and can be used for explicit synchronization, with the guarantee that all memory referenced by the task is consistent.

Once data are transferred, the values of the kernels arguments are set and the kernels are scheduled for execution. Once the last kernel execution is completed, the output memory is transferred to the host issuing the *clEnqueueReadBuffer* command.

As final step, the results of each iteration coming from each thread, are summed up channel by channel to get the *integrated* spectral power.

7.3.1 The kernels functions

Each iteration activates the GPU kernels listed here:

reader kernel: the data copied from host memory are converted to single floating point format, and convolved with the windowing function, whose values have been previously stored into the device constant memory.

radix-n FFT kernel: the data elaborated by the previous kernel are partially transformed to frequency domain. To get the Fast Fourier Transform of the input, this kernel is called a number of times which is a function of block's length and of the FFT radix. The FFT kernel operates on different input and output buffers. In the successive iteration is then called with input and output buffers swapped.

writer kernel: the modulus of each spectra computed by the *FFT kernel*, is averaged, channel by channel, and then transferred back to the memory host. The output of this kernel has a size much smaller than the input of the first one, obtaining a memory bandwidth improvement.

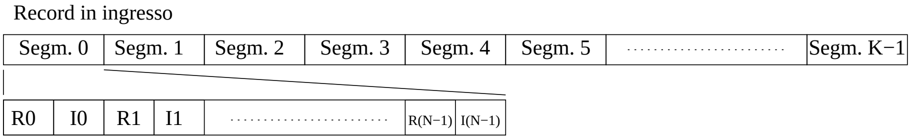

Figure 3: Input data format

7.4 `SpectralGpu` memory synchronization

The synchronization of all the running tasks is under the host control since any type of communication between the threads, or commands devices, takes place through the host memory buffers.

The `gpu threads` share the same host memory buffers for input and output data access. For each thread running, the data used by the computational kernels is first transferred from the global input host buffer *scr_ptr* to the input device buffer allocated on the specific context (*src_buf*).

At the end of computation, the specific output device buffer *res_buf* is moved back to the global output host buffer *res_ptr*.

The `SpectralGpu` host program handles the synchronization between the threads running on the CPU cores using standard objects like semaphores or mutexs, and with the GPU memory commands through the OpenCL event objects.

7.4.1 The host and device memory buffers and how synchroniziation really happens

The input global host memory buffer *src_ptr* is a dynamic array of 4 blocks of memory (see § 7.1) allocated during the program start-up phase.

The buffer *src_ptr* is managed like a FIFO: two semaphores *empty* and *full* control the number of memory blocks available for write or read operations. The first semaphore is initialized to 4, the array size, the second to 0.

Also the output global host buffer *res_ptr* is a dynamic array composed of several blocks, whose length is equal to the number *gpu_threads* of GPU threads active on the system.

Each *GPU thread* is identified by an integer number $n = 0, 1.., gpu\_threads - 1$ and this `id` is used to address the *res_ptr* block of data. This way the *GPU thread* 0 is linked to the first block of *res_ptr*, the *GPU thread* 1 to the second and so on. Since each thread has its own output buffer, no mutex is needed to access it.

Once in execution, each

gthread tries to acquire a *full* semaphore and it blocks if there is no chunk of data available.

The `reader thread`, on the contrary, acquires an *empty* semaphore at start, stores the data received from the network (or the file) directly inside the first available chunk of the input host buffer, and when done, releases the semaphore now *full* to signal the consumer thread, i.e. the gpu one, that new data are ready.

At this point one of the gpu threads waiting on the *full* semaphore can acquire it, blocks the mutex controlling the access to FIFO and begins copying the data to the input device buffer.

The *GPU thread* releases the mutex only when the OpenCL event coupled with the memory write operation, signals the command completion. This is a synchronization point between the *GPU thread* running on the host side and the commands in execution on the device.

Only after this step, the buffers in host memory can be reused by the `reader` and *GPU thread*s for ext iteration: the *GPU thread* releases the *empty* semaphore to signal the `reader thread` that the input data block is available again to be filled with new data. Now the first kernel function, the *reader* one, can be enqueued to start GPU computations.

At the end of the execution of the last kernel, the *writer kernel*, the data stored into the device memory buffer has to be copied back to the global output host memory to finalize the averaging process: each *GPU thread*, identified by the number `id` copies the content of the device memory buffer *res_buf* into the corresponding block *res_ptr[id]* of the output buffer. This operation is controlled through an event object, too.

8 Test and conclusions

To get an high input bandwidth to simulate the working case, we have created a ramdisk and copied in it the 2GB input data file.[3]

To read a block of data of 32 MB (= 32Ksample * 512 channels * 2 bytes) the application

[3] The spcgpu application reads the input data file using fixed-size blocks of 32 MB each, so the program accesses only 1.86 GB of the 2 GB available. This is the value to be used in bandwidth calculation.

takes a mean time of 7 msec, equivalent to an input bandwith of 30 Gbit/sec, higher than the specification maximum required to the project.

To get different input bandwidth, after each data reading operation we have added a time delay whose value is configurable at program start-up[4]

The program supports three different execution mode:

- one GPU with only one *GPU thread* running
- one GPU with two *GPU thread*s running[5]
- two GPUs with one *GPU thread*running each one[6]

Tuning the reading delay, we can estimate the maximum bandwitdh supported in each case using a FIFO depth of 128 MB (4 chunk * 32 MB).

We have executed all the tests by repeating the same read-compute loop for 50 times. For each execution case we have adjusted the input delay to not incur into a FIFO overflow error.

To get more data, and improve its statistic, it's essential to implement the GPU fan control since during the tests the temperature rises steadily and running the tests for longer times can damage the video devices.

In the first execution case, we needed to implement a minimum delay of nearly 28 ms to not get FIFO overflow.

The whole test gives as results a global execution time of 210 sec.

Using the formula :

$$bandwith = \frac{8 \cdot N \cdot nloop}{exec\_time} Gbit/sec$$

where $N = 1.86$, $nloop = 50$ and $exec\_time = 210$ we get a mean input bandwidth of 3.57 Gbit/sec.

In the second case, we got an execution time of 181 sec, corresponding to an input bandwith of 4.1 Gbit/sec.

[4] The argument option *-d* specifies the number of 5ms delay unit.

[5] Only for Capeverde GPU

[6] The Juniper GPU has not enough resources available to run two threads.

The last test, involving the use of two GPUs, we gets as final result a maximum bandwidth of 5.2 Gbit/sec.

If we use 512 channels instead of the default 256[7], we get greater bandwidth values at the cost of more allocated memory for the FIFO (256 MB). In this last case we obtain, for the same type of tests, the values: 3.92 Gbit/s, 4.4 Gbit/s and 5.93 Gbist/s.

At a preliminary analysis we see that passing from 1 to 2 threads per GPU we get a performance increase of 15%, while passing from 1 to 2 GPUs we get an increase of 50%.

[7] the number of channels used can be configured through the-n option

References

[1] “Fast 10GHz Ethernet Driver”, G. Comore, in preparation.

[2] “The Cray-1 Computer System”, 1976, Cray Research Inc.

[3] “The Cray-1 Computer System”, R.M. Russell, Comm. ACM, Jan. 1978, pp. 6372.

Contents