Implementation of a multi GPU version of the cuInspiral pipeline

Loriano Storchi, Leone B. Bosi

19 maggio 2011

1 Introduction

Moore's Law is a violation of Murphy's Law. Everything gets better and better this is how Gordon Moore commented the law, that bears his name, in 2005. Gordon E. Moore formulated the law by a simple observation. In 1965 he noted that number of components in integrated circuits had doubled every two years from the invention of the integrated circuit in 1958. Thus, he predicted that the trend would continue for at least ten years. Some years after the law was reformulated by taking into account an higher growth, the final formulation stated that integrated circuits would double in performance every 18 months.

Along last 20 years hardware manufacturers introduced several architecture innovation to maintain the trend dictated by Moore's law, that is now used in the semiconductor industry to guide long-term planning and to set targets for research and development. Architectural innovations have gone in the direction of introducing implicit and explicit parallelization concepts, and this has been used as a way to go around the obvious miniaturization limitations and frequency increment.

Starting from 2005 multi-core CPU have been introduced in the everyday computing architecture, both in the embedded and standard systems. This solution implements multiprocessing in a single physical package, namely the full processor, replicating the whole computing core. The actual multi-core CPU implements up to four/six cores per package. In case of the multi-core CPU the performance gain is strictly related to the quality of the parallelized software. Referring to this concept we have to quote Amdahl's law, that connect the parallelization gain with the fraction of the software that can be parallelized in order to run on multiple cores simultaneously. The next obvious step in this direction is the many-core architecture. Thus, computing units where several tens of cores are connected together. The actual state of art in many-core architecture is represented by GPU processors, where hundreds of computing cores are implemented within a single package.

2 The cuInspiral prototype pipeline

The cuInspiral prototype pipeline has been developed using CUDA, the computing engine of the NVIDIA Processors. The CUDA execution model is well known, and it can be briefly summarized as a SIMD, or SIMT (Single Instruction Multiple Thread), model, where each "kernel" is executed N times in parallel by N different CUDA threads into a GPU multi-core processors. In the CUDA framework the programmer can define some spacial "functions" called "kernel", that are executed by the GPU. In a simplified vision it can be quoted that the GPU works as a co-processor respect to the host computer, executing the various kernel functions.

The pipeline can be briefly summarized as follow:

- Template generator: the first step is the generation of a template, starting from star masses (m1,m2).
- Matched Filtering: the matched filtering formula is applied in fre quency domain between each polarization and the input signal.
- DFT: an inverse DFT is applied on each previous polarization correlator output, in order to return in time domain.
- Correlator: the two polarization correlators are combines together. The output is a time domain vector, called "correlator output".
- Find Max: we search for "correlator output" values above threshold, than could be associated with true detection.

all the above described operations are performed using not only a CPU but also the GPU. We firstly re-engineerized the code to be used as a library containing functions for the CB (Coalescing binary) detection pipeline fully working on GPU device. After, using the cuInspiral software library, we implemented a program called signaldetection that performs the detection of coalescing binaries (CB) gravitational wave signal using a single GPU.

The workflow of signaldetection results to be simple. There are two main input files, signal file and masses file, and two corresponding nested loops, the first loop iterate over the input signal. So the program reads a data chuck after the other until the is no more input signal. While the inner loop iterates over an input list of star masses, every time a pair of masses has been read the previously described pipeline is called. The output of signaldetection is a series of correlators. Figure 1 summarize the described workflow.

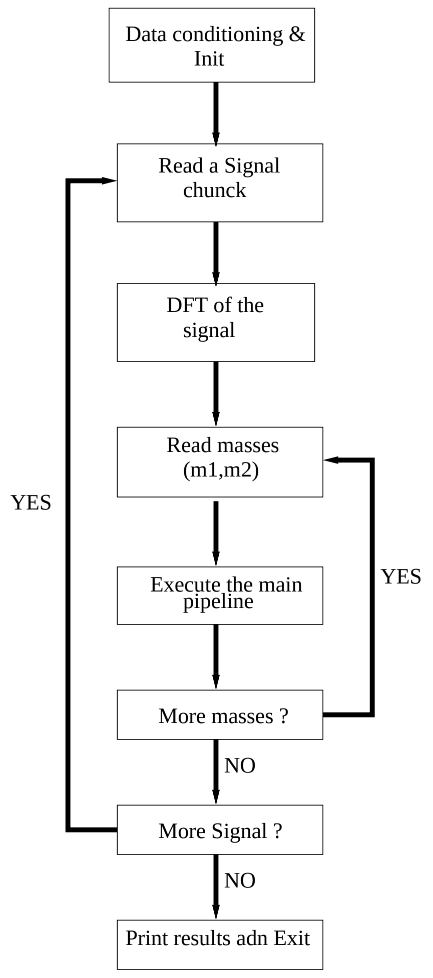

Figura 1: Signaldetection workflow.

The next obvious step is the multi-GPU parallelization. We will discuss this topic in the next section.

3 Multi-GPU parallelization scheme

We adopted two different parallelization schema, the first can be classified as a "master slave" and the second is a pure "data parallel". The implementation of the two approaches as been made using Posix Thread in the first case, and MPI in the latter.

The first approach can be briefly summarized by the following C psuedo-code:

```
main()
{
  for (i=0; i<num_of_thread; i++)
  {
    inititialize (thread_data[i]);
    create_thread (thread_id[i], slave_thread, thread_data[i]);
  }

  inititialize (reader_thread_data);
  create_thread (thread_id[num_of_thread], reader_thread,
                 reader_thread_data);

  do
  {
    correlator_barrier;

    if (nomorecorrelator)
      break;

    reduce (correlator);

    print (correlator);
  }
  while (!end_of_signal)

  for (i=0; i<num_of_thread; i++)
    pthread_join (thread_id[i]);

  pthread_join (thread_id[num_of_thread]);
}

slave_thread (thread_data)
{
  cuda_set_device(thread_data.devicenumber);
  init_and_allocate_data;

  while(1)
  {
    signal_is_ready_barrier;

    if (finalize)
      break;

    copy_signal_from_CPU_to_GPU;

    signal_moved_to_gpu_barrier;

    fft_on_GPU (signal);

    for (i=threadid; i<dim; i+=num_of_thread)
    {
      generate_template;
      matched_filter;
      inverse_FFT;
      combines_correlators_together;
      find_max;
    }

    transfer_data_GPU_to_CPU;

    correlator_barrier;
  }

  nomorecorrelator = true;

  correlator_barrier;

  deallocate_data
}

reader_thread (reader_thread_data)
{
  open (signal_file);
  read (signal);

  signal_is_ready_barrier;

  do
  {
    signal_moved_to_gpu_barrier;

    read(signal);

    if (end_of_file)
      finalize = 1;

    signal_is_ready_barrier;
  } while (end_of_file)
}
```

It come straightforward to note that there are several threads involved in the computation: *main_thread*, the *reader_thread* and finally several *slave_thread*. The *main_thread* is the one that creates the other threads and coordinates the work. The *reader_thread*, the name is quite self-explicative, reads the input signal. Finally each *slave_thread* performs the main computation, each one using a different GPU.

The whole procedure can be briefly described as follows. The *main_thread* creates N + 1 threads, where N is the number of GPU we want to use. After this initialization step it only needs to synchronize with all the *slave_thread* using the *correlator_barrier*, any time the results are ready to be processed. Thus, the *main_thread* performs a reduction in order to find the max values between all the computed correlators.

The *reader_thread* is the thread reading the input signal step by step. It works under the condition of a couple of barriers, namely the *signal_is_ready_barrier* and the *signal_moved_to_gpu_barrier*. As soon as the signal has been read the thread synchronizes with all the slaves using the *signal_is_ready_barrier*. After this barrier the slaves start moving the signal from the CPU to the GPU, when the data transfer is completed a new synchronization occurs through *signal_moved_to_gpu_barrier*. After the second barrier the *reader_thread* can start reading an extra slice of the signal, and the slaves can start the main computation.

Each slave, or worker, thread needs to get synchronized with both the main and the reader thread. Each slave calls a barrier at the beginning of the computation to read the signal, and a barrier at the end of the computation to move the results from the GPU to the CPU. It is quite obvious that each slave is free to perform the computation while the *reader_thread* reads the signal. Besides that, each slave performs the matched filter procedure on a different GPU using a subset from the complete template set. This process is enough to distribute the total computational burden between the different GPU.

3.1 Executing of FFT on the CPU

A possible next step in the direction of overlapping GPU and CPU computation time, is to let the CPU performing the FFT of the input signal. This step induces a minor modification to the code, so that the *fft_on_GPU (signal)* operations are no loger performed and instead the FFT is performed by the *reader_thread* immediatly after the input signal has been read. We may expect that this new approach will become favorable as soon as we have enough star masses, so that the time needed by the GPU to execute the main pipeline is comparable with the one needed by the CPU to perform the FFT.

In any case we can take advantage of many core CPU using the multi-threaded version of the FFTW. In order to be sure to use the best possible configuration for each size of the data chuck, we performed some tests. Some of them used a different number of threads and was observed, as it is well know, that in general the use of a number of threads greater than the number of physical cores is a good choice. Thus, we firstly optimized the number of threads to use given the FFT size, and afterwards we used this number of threads to run the whole code. The results obtained are preliminary but overall satisfactory.

3.2 Multi-GPU using MPI

Finally we adopted a different scheme using MPI instead of POSIX Thread. This new approach can be briefly summarized by the following C psuedo-code:

```
int main (int argc, char *argv[])
{
  int rank, size;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);
  MPI_Status status;

  cudaSetDevice (rank);
  init_and_allocate_date;

  read(signal);

  do
  {
    FFT_on_GPU(signal);

    for (int i = rank; i < masses_dim; i += size)
    {
      generate_template;
      matched_filter;
      inverse_FFT;
      combines_correlators_together;
      find_max;
    }

    transfer_data_GPU_to_CPU;

    MPI_Barrier(MPI_COMM_WORLD);

    MPI_Reuuce;

    if (rank == 0)
      print_results;

    read(signal);

  }
  while (EOF_of_signal);

  free_and_finalize;

  MPI_Finalize ();
}
```

The workflow is much simpler compared to the previously discussed approach. In fact, in this case we run as many processes as GPU we want to use. For instance, if we want to perform our computation using N GPU we run the same number N of MPI processes. Each process will run the main pipeline on a subset of the total amount of star masses. At the end of the computation a reduction of the computed correlators is needed to obtain the final results to be printed.

It is important to stress that the main goal of this approach is the use of MPI, fot gaining the possibility of running the same code in a cluster of GPU. Thus, by having a typical cluster of workstations with several nodes, and each node with one or more GPU, we can run our MPI code as it is.

3.3 Preliminary results

In this section some preliminary results obtained by running our code using up to four NVIDIA Tesla C1060 Computing Processors are reported. The GPUs were installed on an host computer running two Intel Xeon CPU E5520 2.27GHz, thus eight physical cores.

We performed in first place some numerical simulations for testing the correctness of the entire processes. We did implement our code in order to use single precision arithmetic, instead of double precision one. This has been done in order to exploit the full power of the GPU. It is well known that starting from the NVIDIA GT200 GPU the hardware has been equipped also with double precision units. But still the peak performance of the double precision units it is lower than the single precision units. By obtaining numerical results, it was showed that only few percentage points in accuracy were lost.

In Table 3.3 we report the results obtained using different number of star masses (i.e. templates), and an input data chuck of 2^{19} points. The results are reported using the three different approaches implemented, thus:

- **fft_GPU**: is the firstly described POSIX thread implementation, where the FFT is executed by the GPU.
- **fft_CPU**: the second implementation described, where the FFT is executed by the CPU.
- **MPI**: the MPI implementation.

In Table 3.3 we report the computation time for the execution of signal-detection with a single GPU, and the time for the execution of the described implementations using 2, 3, or 4 GPUs. While in Table 3.3 we report the same results above but using an input data chunk of 2^{23} points.

Overall the results obtained show a number of things. Firstly, the MPI implementation has to be chosen especially when using more star masses.

It seems to be favorable probabbly due to the lower complexity of the communications. Indeed in the case of POSIX thread the presence of multiple barriers appears to affect the performances. The other interesting consideration is that, when using more star masses, thus more templates, the implementation executing the FFT on the CPU is favorable. This results was certainly expected and confirms the good overlap between CPU and GPU computation.

3.4 Conclusions

The most plausible computing scenario of the near future is a combination of CPU and GPU technologies, as an evolution of most recent AMD's APU or Intel's Knight ferry technologies. We already showed that it is possible to exploit GPU power in Coalescing Binaries Detection. This means that manycore programming will not be a choice, because it will be status-of-art of near future.

In order to understand the real capabilities of these new architectures, it is fudamental to hilight the obtained results. Our tests, based on a fully Multi-GPU implementation of a Coalescing Binaries Detection pipeline, that includes specifically an input data conditioning, signal Post Newtonian generator up to PN 3.5 and a complete matched fitlering procedure with colored noise.

Compared to CPU implementation of the same algorithms, results show an average gain factor (normalized by price) of about 50, using a single C1060 GPU. This can be translated in a number of applyied matched filtering per seconds of about **30**. Obviously this number depends on vector size. This numbers are about length of $2^2 3$ samples. Using shorter vector it is possible to achieve higher gains. We performed the same test with the new NVIDIA Fermi GPU (i.e. the Tesla C2050), it resulted that this number increase up to **120 templates per second**. The Multi-GPU version of the pipeline, which gives another increasing factor of 3.5 using 4 GPU, led us to an impressive result of about **400 templates per second** processed.

4 Bibliography

- Leone B. Bosi, Loriano Storchi, "Impact of GPU Technology on gravitational wave physics and signal detection systems", E4 Workshop 2010, Bologna, Italy, 16-17 September 2010.

- "Einstein gravitational wave Telescope conceptual design study", ET-0106A-10, S Aoudia, P Amaro-Seoane, F Barone, L Bosi, S Braccini,

(a) **Number of masses : 120**

Single GPU	5.87 s		
GPU	**2**	**3**	**4**
fft_GPU	2.98 s	3.06 s	2.97 s
fft_CPU	2.99 s	2.52 s	2.50 s
MPI	3.94 s	3.41 s	3.20 s

(b) **Number of masses : 240**

Single GPU	9.47 s		
GPU	**2**	**3**	**4**
fft_GPU	5.09 s	3.99 s	3.59 s
fft_CPU	5.00 s	4.01 s	3.80 s
MPI	5.74 s	4.65 s	4.21 s

(c) **Number of masses : 480**

Single GPU	17.03 s		
GPU	**2**	**3**	**4**
fft_GPU	8.68 s	6.73 s	5.79 s
fft_CPU	9.04 s	6.85 s	6.09 s
MPI	9.57 s	7.15 s	5.97 s

(d) **Number of masses : 960**

Single GPU	32.28 s		
GPU	**2**	**3**	**4**
fft_GPU	17.09 s	12.37 s	11.11 s
fft_CPU	16.50 s	12.72 s	10.82 s
MPI	16.41 s	14.32 s	10.32 s

(e) **Number of masses : 1920**

Single GPU	63.68 s		
GPU	**2**	**3**	**4**
fft_GPU	32.95 s	24.70 s	20.79 s
fft_CPU	33.37 s	24.81 s	20.84 s
MPI	30.69 s	22.39 s	19.93 s

(f) **Number of masses : 3840**

Single GPU	130.40 s		
GPU	**2**	**3**	**4**
fft_GPU	66.50 s	52.11 s	44.00 s
fft_CPU	69.87 s	52.82 s	46.61 s
MPI	61.33 s	45.36 s	40.99 s

(g) **Number of masses : 7680**

Single GPU	283.29 s		
GPU	**2**	**3**	**4**
fft_GPU	162.38 s	121.27 s	106.92 s
fft_CPU	180.76 s	124.92 s	105.68 s
MPI	157.07 s	112.14 s	91.32 s

(h) **Number of masses : 15360**

Single GPU	958.30 s		
GPU	**2**	**3**	**4**
fft_GPU	546.12 s	361.42 s	306.05 s
fft_CPU	558.06 s	380.86 s	301.72 s
MPI	569.44 s	347.19 s	258.96 s

Tabella 1: Speedup using an input data chuck of 2^{19} elements

(a) **Number of masses : 120**

Seriale	90.17 s		
GPU	**2**	**3**	**4**
fft_GPU	53.76 s	48.87 s	38.83 s
fft_CPU	45.16 s	40.10 s	45.92 s
MPI	61.83 s	54.95 s	49.73 s

(b) **Number of masses : 240**

Seriale	145.75 s		
GPU	**2**	**3**	**4**
fft_GPU	72.82 s	54.74 s	45.73 s
fft_CPU	73.05 s	54.15 s	46.00 s
MPI	89.28 s	71.93 s	63.37 s

(c) **Number of masses : 480**

Seriale	257.38 s		
GPU	**2**	**3**	**4**
fft_GPU	126.96 s	92.03 s	74.61 s
fft_CPU	128.81 s	93.00 s	75.50 s
MPI	144.21 s	107.42 s	91.66 s

(d) **Number of masses : 960**

Seriale	482.47 s		
GPU	**2**	**3**	**4**
fft_GPU	237.16 s	166.71 s	131.77 s
fft_CPU	241.06 s	168.85 s	132.20 s
MPI	254.19 s	182.59 s	144.93 s

(e) **Number of masses : 1920**

Seriale	932.53 s		
GPU	**2**	**3**	**4**
fft_GPU	457.13 s	318.00 s	246.48 s
fft_CPU	464.82 s	321.03 s	245.65 s
MPI	470.73 s	331.84 s	275.17 s

(f) **Number of masses : 3840**

Seriale	1835.18 s		
GPU	**2**	**3**	**4**
fft_GPU	901.21 s	624.99 s	s 478.74 s
fft_CPU	915.03 s	627.08 s	s 477.75 s
MPI	913.53 s	629.43 s	s 485.94 s

(g) **Number of masses : 7680**

Seriale	3651.28 s		
GPU	**2**	**3**	**4**
fft_GPU	1814.03 s	1259.75 s	956.18 s
fft_CPU	1859.06 s	1235.98 s	954.73 s
MPI	1832.76 s	1258.90 s	959.85 s

(h) **Number of masses : 15360**

Seriale	7578.05 s		
GPU	**2**	**3**	**4**
fft_GPU	3839.54 s	2590.53 s	2011.64 s
fft_CPU	3850.16 s	2596.82 s	2009.53 s
MPI	3719.47 s	2505.81 s	1895.30 s

Tabella 2: Speedup using an input data chuck of 2^{23} elements

C Bradaschia, J van den Brand, C Van Den Broeck, G Cella, J Colas, K Danzmann, T Dent, R De Rosa, V Fafone, P Falferi, R Flaminio, J Franc, F Frasconi, A Freise, D Friedrich, G Gemme, E Genin, C Gräf, S Hild, K Kokeyama, B Krishnan, M Lorenzini, H Lück, E Majorana, M Mantovani, B Mours, H Müller-Ebhardt, R Nawrodt, G Parguezt, A Pasqualetti, M Punturo, P Puppo, D Rabeling, T Regimbau, S Reid, F Ricci, A Rocchi, S Rowan, L Santamaría B Sathyaprakash, L Storchi, S Tarabrin, A Thüring, P Weßels

- Matteo De Bonis, "Pseudo-Random number generators on GPU", Universitá degli Studi di Perugia Facoltá di Scienze MM.FF.NN. Corso di Laurea in Informatica, Relatori: Leonello Servoli, Loriano Storchi, Leone Bosi

- Lanfranco Fontana,"Using MPI and Pthreads in the parallelization of multi-GPU algorithms for the identification of gravitational signals", Universitá degli Studi di Perugia Facoltá di Scienze MM.FF.NN. Corso di Laurea in Informatica, Relatori: Leonello Servoli, Loriano Storchi, Leone Bosi