Parameter optimization for joint source-channel coding

Sergio Pino
Department of Electrical and Computer
Engineering, University of Delaware
140 Evans Hall, Newark, DE
sergiop@udel.edu

ABSTRACT
Joint source-channel coding (JSCC) has been introduced as an alternative to the traditional transmission of analog signals, such as images and video, over wireless channels. JSCC systems have shown not only near-optimum performance for high data rates, but also the ability to cope with varying channel qualities at a very low complexity. A straightforward implementation of JSCC is implemented using the so called Shannon-Kotel'nikov mappings. However, in order to cope with the different channel qualities the parameters of the mappings should be optimized. This paper explores a GPU implementation of the naïve optimization algorithm. We report a speedup up to 19x over the serial version of the algorithm.

Keywords
Joint source-channel coding, Shannon-Kotel'nikov mappings, GPU computing, algorithms, profiling, parallel programming.

1. INTRODUCTION

In 1948 Shannon introduced the idea that the fundamental problem of communication is that of reproducing at one point either exactly or approximately a message selected at another point. As a result, most communication systems today use separate source and channel coders. Currently the transmission of analog source signals such as images, sound, and video use this approach to convey such signals from one point to another.

Recent studies have suggested that under some conditions Joint source-channel coding (JSCC) systems might both perform more robustly in varying channel conditions and perform better in higher spectral efficiency . In fact, a JSCC system might benefit from allowing the channel noise to introduce errors into the transmitted source signal. Interestingly, direct transmission of uncoded Gaussian samples transmitted over an additive white Gaussian channel is optimal. In other words, Gaussian sources match perfectly to Gaussian channels. Previous work has investigated possible schemes, based on analog transformations, to exploit the idea that a perfect match of source and channel distributions yields in optimality.

Within the practical analog coding schemes that have appeared in the literature, those based on the use of Shannon-Kotel'nikov mappings, already proposed more than 50 years ago by Shannon [1] and Kotel'nikov [2], have recently acquired a renewed importance due to the work of Chung [3], Ramstad [4] and Hekland [5]. In this scheme, the encoding idea to reduce the number of samples to be transmitted is to represent a tuple of n source samples as a point in a n-dimensional space where a space-filling surface of dimension k lives. Then, the n-tuple is projected onto the curve and the corresponding k-tuple is transmitted through the noisy channel. In this work we focus on the cases where the dimension of the source is greater than the dimension of the channel $n>k$. This can be interpreted as a lossy compression. Finally, Maximum Likelihood (ML) or Minimum Mean Square Error (MMSE) decoding is performed to recover the original data. However, an optimization needs to be performed for all parameters (Δ and α in spiral-like curves) and for each one of the values of the signal-to-noise ratio (SNR). The optimization is done numerically by calculating through simulation the output SDR for different (Δ, α) pairs.

This work explores a GPU parallel implementation of the naive optimization of the parameters Δ and α, for each one of the values of the SNR. Finally, the parameter optimization of a 2:1 system, (n, k) = (2, 1), is developed and presented to illustrate the speedups attained by a

parallel implementation when compared with the serial version of the algorithm. We report a speedup up to 19x over the serial version of the algorithm.

The remainder of this paper is organized as follows. In the next section, we introduce the serial version of the optimization algorithm for the 2:1 system using Shannon-Kotel'nikov mappings. Section 3 presents the tow parallel versions that target the OpenCL language. Section 4 presents the comparison results in terms of speedup. Finally, conclusions are drawn in Section 5.

2. SERIAL IMPLEMENTATION OF THE OPTIMIZATION ALGORITHM

Since an optimization needs to be performed for all parameters (Δ and α in spiral-like curves) and for each one of the values of the signal-to-noise ratio (SNR). In addition, this optimization is done numerically by calculating through simulation the output SDR for different (Δ, α) pairs. An example of how such a problem looks like is shown in fig 1.

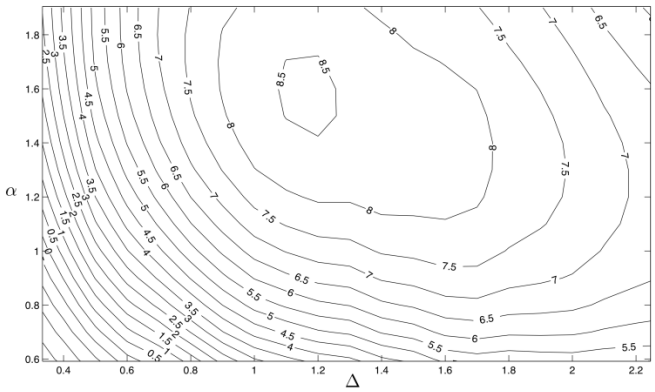

Figure 1: Signal-to-distortion ratio as a function of parameters Δ and α in the spiral-like curve for a Gaussian source when ML decoding is used and the channel SNR = 20 dB.

The idea of the optimization algorithm is to create a grid of pair (Δ, α) values and then execute a simulation of the communication process, using Shannon-Kotel'nikov mappings, for a given quality of the channel or SNR. Then, the simulation computes the signal-to-distortion ratio in order to measures the quality of the reconstructed signal with respect to the original signal.

The system performance is measured in terms of the output signal-to-distortion ratio (SDR) versus the SNR of the channel. The SDR, in dB, is defined as

$$SDR = 10\log_{10}\left(E[\mathbf{x}^2]/_{MSE}\right), \qquad (1)$$

where $\mathbf{x} \in \mathbf{R}^N$ is the original message that was transmitted. The quantity $E[\mathbf{x}^2]$ is the mean of the square of the values of a vector computed as $E[\mathbf{x}^2] = \frac{1}{N}\sum_{i=1}^{N} {x_i}^2$. The distortion between the reconstructed source vector $\hat{\mathbf{x}} \in \mathbf{R}^{\mathbf{M}}$ an the original source vector $\mathbf{x} \in \mathbf{R}^{\mathbf{M}}$ is computed using the mean square error defined as $MSE = \frac{1}{N}\sum_{i=1}^{N}(x_i - \hat{x}_i)^2$. The SNR can be computed as $SNR = 10\log_{10} {}^{1}/_{\sigma_n^2}$, where σ_n^2 is the variance of the noise.

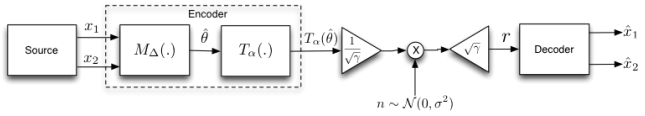

Figure 2: Block diagram of the 2:1 system based on Shannon-Kotel'nikov mappings.

The simulation of the encoding and decoding system has three main steps that have to be performed. In fig. 2, it shows the required steps to perform a simulation. First, the message (a vector with the values of an image) to be transmitted has to be read or generated. Next, the encoding part performs a computation where per each two values of the source (x_1, x_2) the encoder will produce just one value $T_\alpha(\hat{\theta})/\sqrt{\gamma}$.

The second step is to pass the encoded signal through the channel. Thus, the resulting vector (which has half of the number of values of the source) is added with samples of the noise (Gaussian distributed) that corresponds to the desired SNR. Finally, the decoder takes one

received sample and generates two reconstructed source samples.

The code that performs the serial simulations is presented in fig. 3. The functions encoder_21(…), channel(…), and ml_decoder_21(…) have internal loops that can be parallelized fig. 4-6.

```
for (j = 0; j < n_alpha; j++) {
    for (k = 0; k < n_delta; k++) {
        // encoding 2:1
        encoder_21(x_ptr, s_ptr, &s_pw, delta_ptr[k],
 alpha_ptr[j]);
        // channel transmission
        channel(s_ptr, r_ptr, s_pw, n_ptr);
        // 1:2 Non Linear Analog Decoder ML
        ml_mmse = ml_decoder_21(x_ptr, r_ptr, xml_ptr,
 *(delta_ptr + k), *(alpha_ptr + j));
        // SDR
        outputsdr[j][k] = 10 * log10f(x_energ *
 SRC_LENGTH / ml_mmse);
        if (outputsdr[j][k] > max_val ||
                (j_max_val == -1 && k_max_val == -1)) {
            j_max_val = j;
            k_max_val = k;
            max_val = outputsdr[j][k];
        }
    }
}
```

Figure 3: Serial implementation of the optimization.

```
    *s_pw = 0;
    // Mapping the source symbols
    for(int i=0; i < CHN_LENGTH; i++)
    {
        float x1 = *(x_ptr + 2*i);
        float x2 = *(x_ptr + 2*i + 1);

        *(s_ptr + i) = expAlpha(mapping_21(x1, x2, del),
alp);
        *s_pw += *(s_ptr + i) * *(s_ptr + i);
    }
    *s_pw /= CHN_LENGTH;
```

Figure 4: Encoding process.

```
// channel
float s_pw_sqrt = sqrtf(s_pw);
for(int i = 0;i < CHN_LENGTH;i++)
{
    *(r_ptr + i) = *(s_ptr + i) + s_pw_sqrt * *(n_ptr + i);
}
```

Figure 1: Pass throught the channel.

```
for(int i = 0; i < CHN_LENGTH; i++) {
    // ML decoding
    ml_demapping_21(*(r_ptr + i), delta, alpha, &xml_ptr[2*i]);

}
```

Figure 5: Decoding process.

The loop that is more computationally intensive is the encoding process, since it includes several instances of the following instructions:

- If statements
- Float operations: sqrt, floor, atan, cos, sin, pow, abs.

The decoding process is straightforward, it includes some float operations (pow, cos, sin, abs). Finally, the channel process is just one multiplication and addition per value of the vector.

These three processes have to be computed for each pair of alpha and delta values, that are represented by the **for** statements in fig. 3.

3. PARALLEL IMPLEMENTATION.

After analyzing the computation hotspots in the serial version, it was possible to determine that a parallelization of the encoding process will yield in a considerable improvement when compared to the serial version. On the other hand, due to the embarisily parallel nature of the channel process and the decoding process we decided also to parallelize these loops.

Using the data dependency analysis, fig. 7, we could determine that during the process there are not data dependencies between adjacent values. This can be perceived when in the encoding process two source values (0, 1) are encoded in the value (0), while the values (2, 3) are encoded in the value (1). Thus, during the whole process we can expect that there is not going to be any resource sharing between work-items. This makes the parallelization simpler and without synchronization statements.

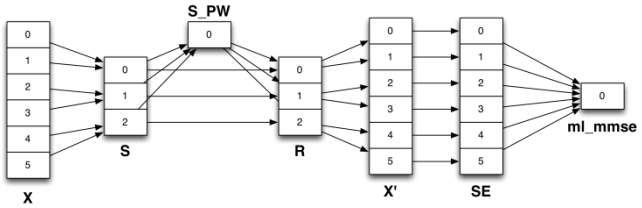

Figure 6: Data dependency for an example when the source has dimension 6 and the resulting encoded vector has dimension 3. X is the source vector, X' is the reconstructed vector.

Due to problems using the reduction algorithm to sum up a vector of numbers entirely in the GPU using OpenCL, these algorithms use the support of the CPU for performing the sum up process. Since between the encoding process and the channel process the value for the power of the vector s (s_pw) has to be computed, and the minimum mean square error (ml_mmse) has to be computed in order to get the performance of the system (SDR).

Thus, the resulting parallel algorithms will deploy two consecutive kernels to the GPU, fig. 8. First, KERNEL_1 (encoder21) performs the encoding of the source vector X into the channel vector S. Then, on the CPU the value of the power of the vector S is computed. Next, the KERNEL_2 (ml_decoder_21) performs in "one step" the channel process and the decoding process. Here, the received vector R is computed in a parallel version of fig. 5, and a parallel version of fig. 6 performs the decoding. In addition, this kernel computes per value the square error.

In OpenCL jargon [6], the parallel program will create a global number of work-items equal to the half of the size of the source vector. Then, these work-items will be divided in groups of size GS which should be a multiple of 32. The kernels in these implementations do not have any synchronization statements.

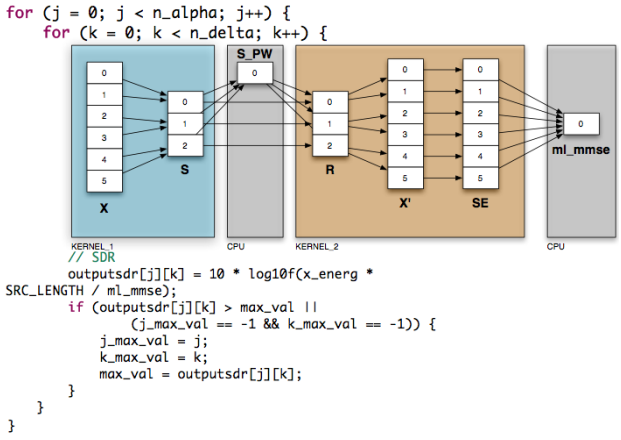

Figure 7: Structure of the parallel implementation.

4. PERFORMANCE RESULTS

The performance improvements expected by using an extremely parallel computer system, as the one provided by a GPU, were investigated using the kernels described in the previous section. These experiments were run on the GPU of the server cuda.acad.ece.udel.edu. The experiments take into account the variations of both the problem size (size of the array used as input in the algorithm) and the work-group size (number of work-items in each work-group). Then, it is interesting to measure the variations in the performance (time) that the GPU implementation achieves when the previous variables are modified.

In order to analyze the changes in the performance of the parallel implementation we use the following values of the work-group size equal to (32, 64, 128, 256, 512) work-items, and the following values for the problem size (2^p) equal to (1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288, 1048576) elements in the source vector.

Figure 9 shows the performance in micro seconds that was obtained using OpenCL. Here, the work-group size was set to the maximum work-group size of the supported by the GPU, so the value was 512 work-items. As expected the serial version grows execution time grows linearly with respect to the problem size. On the other hand, the parallel versions keep more stable and grown slowly. This is due to the highlight parallel nature of the whole encoding and decoding process, even if some intermediate results have to be computed on the CPU. Also, this slow grow shows the computational power of the GPU technology.

Figure 10 shows the speedup of the parallel implementations. Here, it is clear that there is an exponential grow in the speedup for problem sizes less or equal to 2^{18} or 262144 values for the source vector. Then, the speedup grows more slowly and eventually reach a maximum point around 19x. This could be due to two factors. First, there are parts of the simulation that are performed by the CPU (computation of s_pw and ml_mmse) that could be seen as a bottleneck that cause the bound at 19x. Also, since these results have to be computed on the CPU, there is an

extra overhead of data transfer from the GPU to the CPU in order to compute the values. Second, the bound could be the product of reaching the GPU hardware resources.

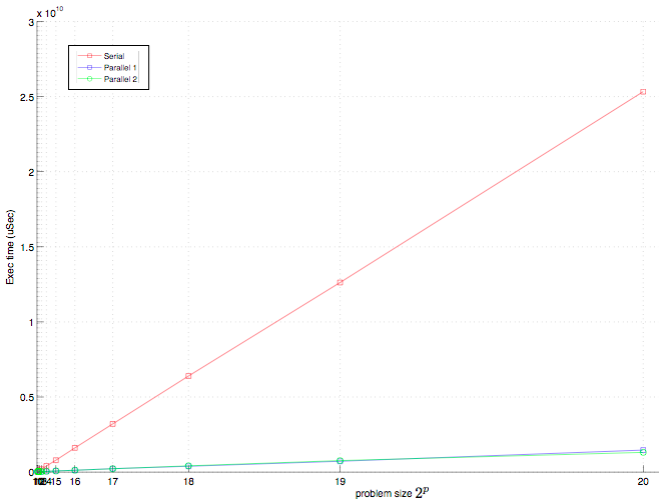

Figure 8: Performance in uSec of the serial and parallel implementations with respect to the problem size.

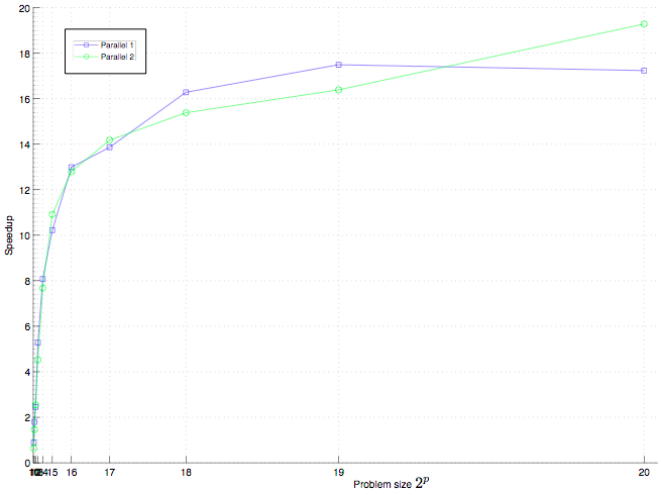

Figure 9: Speedup of the two parallel implementationversioned to the serial versioon.

On the other hand, interestingly we found that a variation of the work-group size has just small variations in terms of the execution time.

5. CONCLUSIONS

In this work we have proposed two approaches for computing, in a parallel fashion using a GPU architecture, the optimization of the parameters for the joint source-channel coding algorithm based on Shannon-Kotel'nikov mappings. We reach up to 19x of speedup when compared with the serial version. However, these implementations have the drawback that uses the CPU for the computation of some intermediate results, which yields in a clear bottleneck in terms of the scalability of the algorithm. Future work could explore the use of the reduction algorithm (for sum up a vector entirely on the GPU) in order to remove the support of the CPU. This could increase the performance since the summation algorithm is highly parallel and the fact of not using the support of the CPU yields in less data transfers between CPU-GPU which is very costly. In this work we explored the use of the reduction algorithm and the MapReduce model, but the efforts were unsuccessful (due to technical problems).

Because predicting the performance effects of program optimizations is difficult, developers or compilers may need to experiment to find the configuration with the best performance. To show this, we developed a testbed application that can be configured to explore different combination of values for work-group size and problem size. These changes help to judge the performance of a parallel configuration. By plotting the configurations and examining the performance results. However, surprisingly in the implemented algorithms the variations were not significant.

The development in OpenCL is a complicated but rewarding experience. One of the main difficulties was the oververbose code imposed by OpenCL, also at the beginning the idea of enqueue several kernels was somewhat complicated to assimilate. However, the fact that OpenCL is not restricted to one platform opens an interesting opportunity to exploit parallelism of different algorithms and integrate in our research activities parallel computing.

REFERENCES

[1] C. E. Shannon, "A mathematical theory of communication," The Bell System Technical Journal, vol. 27, pp. 379-423, 1948.

[2] V. A. Kotel'nikov, "The theory of optimum noise immunity," New York: McGraw-Hill Book Company, Inc, 1959.

[3] S. -Y. Chung, "On the construction of some capacity-approaching coding schemes," Ph.D. dissertation, Dept. EECS, Massachusetts Institute of Technology, 2000.

[4] T. A. Ramstad, "Shannon mappings for robust communication," Telektronikk, vol. 98, no. 1, pp. 114-128, 2002.

[5] F. Hekland, G. E. Oien, and T. A. Ramstad, "Using 2:1 Shannon mapping for joint source- channel coding," Proc. DCC'05, March 2005.

[6] Scarpino, Matthew. "OpenCL in action." (2011).