OPTIMIZATION OF FAST HARTLEY TRANSFORM (FHT) FOR GPU ACCELERATION *

Carlos Osorio Quero
Computer Science Department
Instituto Nacional de Astrofisica Optica y Electronica
Mexico,Puebla
caoq@inaoep.mx

ABSTRACT

The recent advancements in edge computing power are largely attributable to technological innovations that enable accelerators with extensive hardware parallelism. One practical application is in computer imaging (CI), where the use of GPU acceleration plays a crucial role, particularly in reconstructing 2D images through techniques like Single-Pixel Imaging (SPI). In SPI, algorithms such as compressive sensing (CS), deep learning, and Fourier transformation are essential for 2D image reconstruction. These algorithms benefit significantly from parallelism, enhancing performance by optimizing processing times. To fully exploit the GPU's capabilities, strategies such as optimizing memory access, unrolling loops, crafting kernels to minimize operation counts, utilizing asynchronous operations, and maximizing the number of active threads and warps are employed. In various lab scenarios, integrating embedded GPUs becomes essential for algorithmic optimization on SoC-GPUs. This study focuses on the rapid optimization of the fast Harley Transform (FHT) for 2D image reconstruction on Nvidia's Xavier platform. By applying diverse parallelism strategies in PyCUDA, we achieve an approximate acceleration x 3, bringing processing times close to real-time https://github.com/1Px-Vision/Project_HPC.

***K*eywords** GPU (graphics processing unit) · CUDA · optimizing memory access · Single-Pixel Imaging (SPI) · system-on-chip (SoC) · fast Harley Transform (FHT)

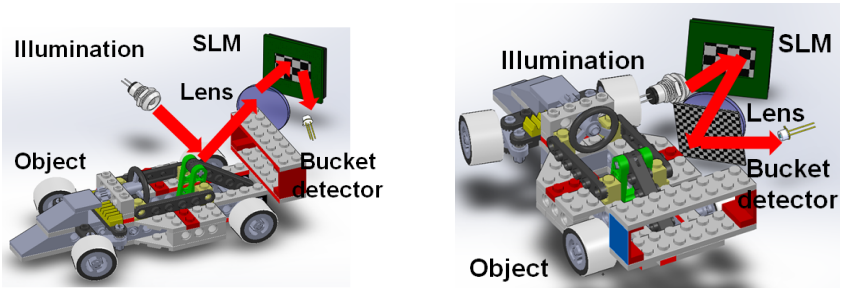
1 Introduction

The Single-Pixel Imaging (SPI) [1] technique represents a cutting-edge method for image reconstruction that requires fewer samples, which are typically sparse in the transform domain, compared to traditional imaging systems. Despite its promise, the signal reconstruction algorithms currently used face challenges, notably due to their heavy computational load and significant power consumption [2]. These issues are particularly critical in applications such as autonomous drone flight or other real-time embedded systems, where rapid processing is imperative. In our research, we explore the potential of a hardware-based parallel processing architecture to execute the Fast Hartley Transform (FHT) algorithm [3, 4], recognized as one of the most effective reconstruction algorithms to date. We leveraged the GPU-accelerated capabilities of the Nvidia Xavier platform, employing PyCUDA [5] to develop kernels aimed at optimizing bottleneck operations. Our optimization strategies included the use of shared memory with padding to prevent bank conflicts, coalescing global memory accesses for increased throughput, pre-computation of constants to diminish runtime calculations, loop unrolling, and warp divergence minimization through conditional optimization [2]. We also selected faster arithmetic operations when the precision requirements permitted and minimized synchronization needs. Our proposed GPU-based implementation of an adapted FHT algorithm focused on 64x64 image reconstruction, utilizing laboratory-derived values. This parallel architecture approach yielded processing times up to x 3 faster than those achieved with conventional CPU-based methods.

* *Citation*: **Carlos Osorio Quero. "Optimization of Fast Hartley Transform (FHT) for GPU Acceleration"**

2 Single-Pixel Imaging

Single-Pixel Imaging (SPI) cameras operate through an intriguing method that shines a sequence of precise light patterns onto a subject. As these patterns are projected, a bucket detector — lacking in spatial resolution — captures the light intensities that bounce back. A key instrument in these cameras is the Spatial Light Modulator (SLM), like the Digital Micro-Mirror Device (DMD), as presented in a specific diagram. There are two primary designs for SPI cameras: one uses structured detection and the other employs structured illumination [1]



(a) (b)

Figure 1: Two different approaches applied to SPI: (a) structured detection and (b) structured illumination [6].

In the process of structured detection, a light source illuminates an object, and the light that bounces back is directed onto a Spatial Light Modulator (SLM). Following this, a bucket detector is used for the detection phase. Structured illumination involves the use of patterned light modulation, known as $\Phi\left(i,j\right)$. These patterns may be in the form of Hadamard sequences, random patterns, or various other types. The light pattern modulates the illumination on the object, referred to as O. The light that reflects off the object is then captured by a bucket detector, which transforms it into an electrical signal represented by y_i. This conversion is expressed in the equation 1 [1].

$$y_i = \alpha \sum_{i=1}^{M} \sum_{j=1}^{N} O\left(i,j\right) \Phi\left(i,j\right) \tag{1}$$

The factor α depends on the optoelectronic response of the photodetector. The image reconstructed x_i from the captured signal y_i and the corresponding pattern Φ represent in the equation (2) [1]:

$$x_i = \alpha \sum_{i=1}^{M} \sum_{j=1}^{N} y_i \Phi\left(i,j\right) \tag{2}$$

3 Fast Hartley Transform (FHT)

The Fast Hartley Transform (FHT) is a real-valued transform that is closely related to the better-known Fast Fourier Transform (FFT) (See Fig. 2) [7]. It offers some advantages over the FFT in certain applications, such as image reconstruction in single-pixel imaging systems. The FHT can be represent as a discrete signal $x[n]$ is defined equation (3):

$$H[n] = \sum_{k=0}^{N-1} x[k] \cdot \left(\cos(2\pi kn/N) + \sin(2\pi kn/N)\right) \tag{3}$$

where N is the total number of samples in the signal, n is the current sample, and k is the running index for the summation. The inverse FHT (IFHT), which is used to reconstruct the signal from its Hartley transform form discrete is defined equation (4)

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} H[k] \cdot \left(\cos(2\pi kn/N) + \sin(2\pi kn/N)\right) \tag{4}$$

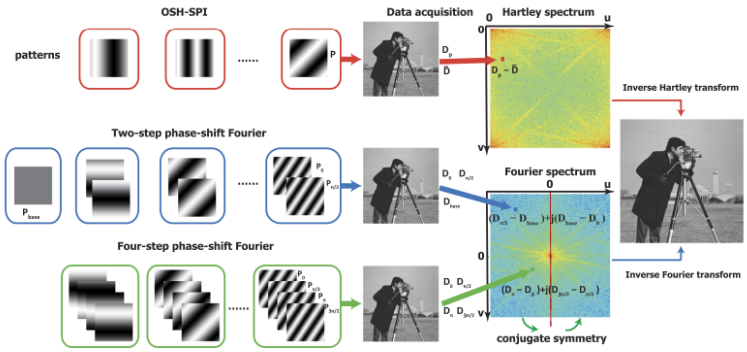


Figure 2: Schematic diagram of the FHT-SPI method, four-step phase-shift Fourier method, and two-step phase-shift Fourier method [7].

In single-pixel imaging, the reconstruction of an image from a series of measurements involves correlating the measured values with the patterns of illumination used to probe the scene. If M patterns are used, the reconstruction can be represented equation (5)

$$I(x,y) = \sum_{m=1}^{M} p_m(x,y) \cdot d[m] \tag{5}$$

where $I(x,y)$ is the intensity of the reconstructed image at the pixel location (x,y), $p_m(x,y)$ represents the m^{th} pattern of illumination, and $d[m]$ is the measurement obtained using the m^{th} pattern. When the FHT is used in this context, each pattern and corresponding measurement are transformed into the Hartley domain. The image reconstruction is performed in this domain, taking advantage of the FHT's properties, and then the IFHT is used to obtain the final spatial domain image.

4 Optimization GPU Fast Hartley Transform (FHT)

The process of reconstructing an image from its spectral data involves using the Inverse FHT (See algorithm 1). This method is efficient when the image dimensions are powers of two, conforming to the FHT algorithm requirements. The reconstruction occurs in two dimensions, applying the Inverse FHT first along the rows, then the columns. Key Functions in the Reconstruction Process:

- **FDHT-CPU Function:** This primary function initiates the image reconstruction process. It requires a data matrix (object_intensity) and a step size parameter (nStep). When nStep is set to 2, the function calculates spectral data (spec) by subtracting the second channel of object_intensity from the first. For execute a 2D Inverse FHT on this spectral data, yielding the reconstructed image (img) (See algorithm 1).
- **FhtseqInv Function:** Serving as the backbone of the Inverse FHT sequence, this function applies the Inverse FHT algorithm to a data vector. It requires the data length to be a power of 2. The function implements the algorithm through bit-reversal ordering and successive updates using a decimation-in-frequency approach (See algorithm 1 lines 6-9).

4.1 Inverse Fast Hartley Transform (IFHT)

The inverse FHT is used to convert data from the frequency domain back to the time domain. This function, specifically, assumes the input data is a sequence that has already been transformed by the FHT and is now being reverted to its original sequence (See Algorithm 2). The steps in the algorithm are descriptor continuation:

- **Input Validation:** The function starts by checking if the length of the input data (N) is a power of 2. This is a common requirement for FFT-like algorithms, as they typically divide the data into halves recursively (See Algorithm 2 line 4).

Algorithm 1 FDHT-CPU Algorithm

1: **procedure** FDHT-CPU($object\_intensity, nStep$)
2: **if** nStep==2 **then** ▷ Check if the step size is 2, indicating a base case for recursion
3: $spec \leftarrow (object\_intensity[:,:,0] - object\_intensity[:,:,1])$ ▷ Calculate the spectrum difference between two consecutive intensities
4: $N \leftarrow len(spec)$ ▷ Determine the length of the spectrum
5: **for** `i to N` **do** ▷ Iterate over each row
6: $spec[i,:] \leftarrow FhtseqInv(spec[i,:])$ ▷ Apply the inverse Fast Hartley Transform to each row
7: **end for**
8: **for** `i to N` **do** ▷ Iterate over each column
9: $spec[:,i] \leftarrow FhtseqInv(spec[:,i])$ ▷ Apply the inverse Fast Hartley Transform to each column
10: **end for**
11: **end if**
12: **return** $spec$ ▷ Return the reconstructed 2D image
13: **end procedure**

- **Bit Reversal Ordering:** The function bitrevorder(data) reorders the data in bit-reversed order, which is a necessary step in many FFT-like algorithms (See Algorithm 2 line 10).
- **Transformation Process:** The outer loop iterates over the stages of the transformation. The two inner loops iterate over parts of the data, applying the inverse Hartley transform calculations (See Algorithm 2 line 11).
- **Inverse Hartley Calculations:** The calculations involve updating pairs of elements in the data array. Depending on the stage of the transformation , the elements are either added or subtracted in a certain way (See Algorithm 2 lines 13-28).
- **Normalization:** The function returns the transformed data, normalized by dividing by N. This normalization step is common in inverse transforms to adjust the amplitude of the signal correctly (See Algorithm 2 line 33).

4.2 Inverse Kernel for Fast Hartley Transform (IKFHT)

To enhance the performance and memory efficiency of a CUDA kernel, we've implemented FHT inverse (See algorithm 3). This approach leverages shared memory and incorporates loop unrolling techniques.These techniques help reduce memory access latency and improve overall computational efficiency by making better use of the GPU's parallel processing capabilities. Below is a comprehensive explanation, complete with detailed comments for improved clarity:

- **Thread and Block Indices:** The variable tid is used to calculate the unique thread ID within the entire grid will be used to access elements in the input array. The variable threadNum represents the total number of threads in the grid (See algorithm 3 lines 2-3).
- **Shared Memory Utilization:** Allocates shared memory dynamically. The size 32 may be indicative of the maximum number of threads in a block or a chosen factor for unrolling (See algorithm 3 line 4).
- **Loop for Processing Each Element:** The outer loop for ensures each thread works on different data points, enabling parallel processing (See algorithm 3 line 10). .
- **Data Loading to Shared Memory:** Each thread loads its element into shared memory, allowing for faster access in subsequent computations (See algorithm 3 line 11).
- **Synchronization Points:** The function syncthreads() is used to synchronize threads within a block, ensuring data integrity in shared memory (See algorithm 3 line 12).
- **Loop Unrolling:** The innermost loop is a candidate for loop unrolling, a common optimization in CUDA to reduce loop overhead and increase parallelism (See algorithm 3 line 15).
- **Transform Computation:** Variables k1, k2, and k3 are used for indexing and are updated in each iteration of the outer loop, which may correspond to different stages of the FHT. The computation involves element-wise operations on the data stored in shared memory, exploiting data locality (See algorithm 3 lines 26-28).
- **Post-Processing:** After the computation, the result is normalized by the size N and written back to the original array (See algorithm 3 line 30).

Algorithm 2 FhtseqInv Algorithm

procedure FHTSEQINV($data$)
  $N \leftarrow length(data)$ ▷ Get the number of data points
  $L \leftarrow log2(N)$ ▷ Determine the number of levels
  if (L- floor(L)) > 0.0) **then**
    ValueError, "Length must be power of 2"
  end if
  $K1 \leftarrow N$ ▷ Initialize K1 as the total number of points
  $K2 \leftarrow 1$ ▷ Set initial value of K2
  $K3 \leftarrow N/2$ ▷ Initialize K3 for the main loop
  $x \leftarrow bitrevorder(data)$ ▷ Reorder data in bit-reversed order
  for `i to L` **do** ▷ Loop over each level
    $L1 \leftarrow 1$
    for `i2 to k2` **do** ▷ Loop for merging subsequences
      for `i3 to k3` **do** ▷ Process each subsequence
        $ii \leftarrow i3 + L1 - 1$
        $jj \leftarrow ii + k3$
        $temp1 \leftarrow x[ii - 1]$
        $temp2 \leftarrow x[jj - 1]$
        if i2 mod 2 == 0 **then** ▷ Conditional operation based on i2
          $x[ii - 1] \leftarrow temp1 - temp2$
          $x[jj - 1] \leftarrow temp1 + temp2$
        else
          $x[ii - 1] \leftarrow temp1 + temp2$
          $x[jj - 1] \leftarrow temp1 - temp2$
        end if
      end for
      $L1 \leftarrow L1 + k1$
    end for
    $K1 \leftarrow round(k1/2)$ ▷ Update K1 for the next level
    $K2 \leftarrow k2 * 2$ ▷ Double K2 for the next subsequence
    $K3 \leftarrow round(k3/2)$ ▷ Halve K3 for the next level
  end for
  return $(1/N) * x$ ▷ Return normalized inverse FHT
end procedure

4.3 Digitrevorder kernel function

The digitrevorderkernel function is designed to perform a digit reversal order computation using a combination of shared memory and loop unrolling techniques (See algorithm 4). This kernel is part of a larger GPU-based program and is structured to efficiently utilize the parallel processing capabilities of NVIDIA GPUs. The steps in the algorithm are descriptor continuation:

- **Thread and Block Indices:** The variable tid is calculation determines the unique index (idx) for each thread across blocks in the GPU grid (See algorithm 4 line 3).
- **Shared Memory:** Shared memory is a faster type of memory accessible by all threads in a block. The size of this array is dynamically allocated based on the value of L, though it's statically defined as 32 here (See algorithm 4 line 5).
- **Loading shared Memory:** The loop `for(i to (threadIdx.x,L), i += blockDim.x)` is designed for coalesced access to global memory (See algorithm 4 line 4). The function syncthreads() ensures that all threads in the block have completed loading data into shared memory before proceeding (See algorithm 4 line 7).
- **Computation:** The #pragma unroll directive instructs the compiler to unroll the loop (See algorithm 4 line 12). Loop unrolling is an optimization that can enhance performance by reducing the overhead of loop control and increasing instruction level parallelism.
- **Writing the Result:** The computed res is stored in the result array at the position corresponding to the thread's idx (See algorithm 4 line 17).

Algorithm 3 FhtseqInv Kernel Algorithm

procedure FHTSEQINVKERNEL($*x, intN, intL$) ▷ Kernel for computing the inverse Fast Hartley Transform.
  $tid \leftarrow blockIdx.x * blockDim.x + threadIdx.x$ ▷ Calculate the thread ID within the grid.
  $threadNum \leftarrow blockDim.x * gridDim.x$ ▷ Calculate the total number of threads in the grid.
  $shared \leftarrow sdata[32 + 1]$ ▷ Allocate shared memory for thread communication.
  $N \leftarrow length(data)$ ▷ Determine the size of the input data.
  $L \leftarrow log2(N)$ ▷ Compute the logarithm base 2 of N for iterations.
  $K1 \leftarrow N$
  $K2 \leftarrow 1$
  $K3 \leftarrow N/2$ ▷ Initialize variables K1, K2, K3 for the loop.
  for `pos to (tid,threadNum)` **do** ▷ Loop over data points assigned to this thread.
    $sdata[threadIdx.x] \leftarrow x[pos]$ ▷ Load element into shared memory.
    $syncthreads()$ ▷ Synchronize threads to ensure data is loaded.
    for `i2 to k2` **do** ▷ Begin inner loop for FHT computation.
      $isOdd \leftarrow i21$ ▷ Check if i2 is odd or even.
      for `i3 to k3` **do** ▷ Iterate over segments of the data.
        $ii \leftarrow i3 + L1 - 1$
        $jj \leftarrow ii + k3$
        $temp1 \leftarrow sdata[ii]$
        $temp2 \leftarrow sdata[jj]$
        $sdata[ii] \leftarrow isOdd?temp1 - temp2 : temp1 + temp2$
        $sdata[jj] \leftarrow isOdd?temp1 + temp2 : temp1 - temp2$

      end for
      $L1 \leftarrow L1 + k1$
    end for
    $K1 \leftarrow >>= 1$
    $K2 \leftarrow <<= 1$
    $K3 \leftarrow >>= 1$
  end for
  $x[pos] \leftarrow sdata[threadIdx.x]/N$ ▷ Write the transformed data back to global memory.
  $syncthreads()$ ▷ Synchronize threads before exiting.
  return x ▷ Return the inverse-transformed 2D array.
end procedure

5 Experimental results

The Fast Discrete Hartley Transform (FDHT) is a crucial algorithm in image processing and computer vision, offering efficient computation for frequency domain analysis. To evaluate its performance, especially when accelerated using a kernel approach, we can analyze the execution times across different hardware configurations (See Tab. 1). To evaluation we used the algorithm proposed in the sections 4. For our experiments, we used the following configuration SoC-GPU Jetson Xavier NX:

- **GPU:** 384-core NVIDIA Volta GPU with 48 Tensor Cores.
- **CPU:** 6-core NVIDIA Carmel ARM v8.2 64-bit CPU, consisting of 6 MB L2 + 4 MB L3 cache.
- **Firware:** JetPack 5.1.2 / Jetson Linux 35.4.1.
- **Linux:** Ubuntu 20.04.

The execution times are measured in milliseconds and are compared across four different configurations: CPU, GPU, Pre-processed (Pre) CPU, and Pre-processed (Pre) GPU (See Fig. 3):

- **GPU:** The execution time on a CPU without any pre-processing is 103 ms. This serves as a baseline for comparison with other configurations.
- **CPU:** When executed on a GPU, the time significantly drops to 45 ms, showcasing the GPU's superior processing power for parallelizable tasks like the FDHT.

Algorithm 4 digitrevorderkernel Algorithm

procedure DIGITREVORDERKERNEL($*x, *vec, *result, N, L, base$)
  $svec[32] \leftarrow shared$ ▷ Allocate shared memory dynamically based on L
  $idx \leftarrow threadIdx.x + blockDim.x * blockIdx.x$ ▷ Calculate global index of the thread
  for `i to (threadIdx.x,L), i += blockDim.x` **do**
    $svec[i] \leftarrow vec[i]$ ▷ Each thread loads a portion of vec into shared memory
  end for
  $syncthreads()$ ▷ Synchronize threads to ensure shared memory is fully loaded
  if idx < N **then**
    $res \leftarrow 0$ ▷ Initialize the result for the current thread
    $temp \leftarrow x[idx]$ ▷ Load the input value for processing
    for `k to (L-1,k >= 0)` **do**
      $\#pragmaunroll$ ▷ Reducing loop overhead
      $div \leftarrow temp/svec[k]$ ▷ Divide temp by current shared memory value
      $res \leftarrow res + div * svec[L-1-k]$ ▷ Accumulate result with weighted sum
      $temp \leftarrow temp - div * svec[k]$ ▷ Update temp for next iteration
    end for
    $result[idx] \leftarrow res$ ▷ Write the computed result back to global memory
  end if
  return $result$▷ Return the final results for all processed elements ▷ This algorithm typically used for 2D image reconstruction
end procedure

- **Pre-processed (Pre) CPU:** Pre-processing the data before running the FDHT on a CPU reduces the execution time to 43 ms. This indicates that pre-indexing and preparing the data can significantly optimize performance even on less powerful hardware like a CPU.
- **Pre-processed (Pre) GPU:** The lowest execution time is observed with pre-processed data on a GPU, at 34 ms. This combination of pre-processing and GPU acceleration yields the best performance, highlighting the effectiveness of optimizing both the data and the hardware used.

Table 1: Evaluating the Performance of the FHT Algorithm: Memory usage, Execution time (ms), and Speedup on CPU vs. GPU.

Hardware	**Memory usage** $\% \downarrow$	**Execution time (ms)** $\downarrow$	**Speedup** $\% \uparrow$
CPU	1.81	103	—-
GPU	1.92	45	x2.28
Pre(CPU)	1.79	43	x2.39
Pre(GPU)	1.24	34	x3

6 Conclusion

We have developed various optimization strategies for the Fast Hartley Transform (FHT) algorithm on the NVIDIA Xavier NX GPU. Specifically, we introduce two types of kernels: one for optimizing the Inverse FHT (IKFHT) calculation (algorithm 3) and another for computing the digit reversal order (algorithm 4). Our evaluation shows significant differences in execution times for the FHT algorithm across different computing platforms. When executed on a Central Processing Unit (CPU), the FHT algorithm takes 103 ms. In contrast, on the GPU, it requires only 45 ms, underscoring the GPU's efficiency, likely due to its advanced parallel processing capabilities, which are ideal for the demands of the FHT algorithm.

Additionally, we have implemented a technique called pre-indexing, which further enhances the FHT algorithm's performance. Pre-indexing involves calculating certain values in advance, which are frequently used during the algorithm's operation. This approach reduces the time required for each iteration. On the CPU, applying pre-indexing nearly halves the execution time to 43 ms, a significant improvement. On the GPU, pre-indexing further reduces the time to 34 ms, indicating that it effectively lightens the computational load, especially in tasks requiring rapid recalculations and image gradient adjustments. Regarding memory usage, it remains low on both platforms, with the GPU having a slightly higher memory usage than the CPU. Introducing pre-processing significantly cuts down memory

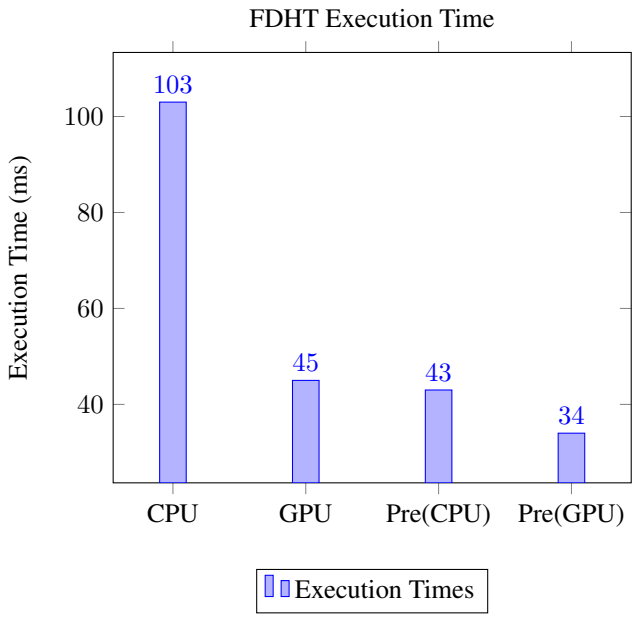


Figure 3: Enhance the FHT 2D reconstruction process by applying various optimization strategies, focusing on CUDA-based parallelization of the algorithms.

usage, especially on the GPU, where it drops from 1.92% to 1.24%. Finally, we calculated the speedup percentage, a crucial metric indicating performance enhancement when using the GPU over the CPU. Without pre-processing, the GPU is approximately 2.28 times faster than the CPU. With pre-processing, this speedup increases to 2.39 times on the CPU and up to 3 times on the GPU, highlighting the benefits of using the GPU and pre-processing techniques for the FHT algorithm.

References

[1] Carlos A. Osorio Quero, Daniel Durini, Jose Rangel-Magdaleno, and Jose Martinez-Carranza. Single-pixel imaging: An overview of different methods to be used for 3D space reconstruction in harsh environments. *Review of Scientific Instruments*, 92(11):111501, 11 2021.

[2] Giridhar Sreenivasa Murthy, Mahesh Ravishankar, Muthu Manikandan Baskaran, and P. Sadayappan. Optimal loop unrolling for gpgpu programs. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–11, 2010.

[3] Mengchao Ma, Qianzhen Sun, Xicheng Gao, Guan Wang, Huaxia Deng, Yi Zhang, Qingtian Guan, and Xiang Zhong. High-efficiency single-pixel imaging using discrete Hartley transform. *AIP Advances*, 11(7):075211, 07 2021.

[4] Anumeena Sorna, Xiaohe Cheng, Eduardo D'Azevedo, Kwai Won, and Stanimire Tomov. Optimizing the fast fourier transform using mixed precision on tensor core hardware. In *2018 IEEE 25th International Conference on High Performance Computing Workshops (HiPCW)*, pages 3–7, 2018.

[5] Nathaniel Bowman, Erin Carrier, and Greg Wolffe. Pygasp: Python-based gpu-accelerated signal processing. In *IEEE International Conference on Electro-Information Technology , EIT 2013*, pages 1–6, 2013.

[6] Carlos Osorio Quero, Daniel Durini, Jose Rangel-Magdaleno, Jose Martinez-Carranza, and Ruben Ramos-Garcia. 3d human pose reconstruction single-pixel imaging. In D. Moormann, editor, *14^{th} annual International Micro Air Vehicle Conference and Competition*, pages 33–39, Aachen, Germany, Sep 2023. Paper no. IMAV2023-4.

[7] Mengchao Ma, Qianzhen Sun, Xicheng Gao, Guan Wang, Huaxia Deng, Yi Zhang, Qingtian Guan, and Xiang Zhong. High-efficiency single-pixel imaging using discrete Hartley transform. *AIP Advances*, 11(7):075211, 07 2021.