MMI713 Term Project: Image Resampling Using Interpolation Methods

1st Tan Çağatay Acar
Electrical and Electronics Engineering
Middle East Technical University
Ankara, Turkey
tacar@metu.edu.tr

***Abstract*—The term project explores the implementation of four image interpolation techniques: Nearest Neighbor, Bilinear, Bicubic, and Lanczos. The objective is to apply these methods to image upscaling tasks and evaluate their performance and output quality on both CPU and GPU platforms using CUDA. Custom codes are compared with OpenCV's existing implementations. Several approaches on memory handling are tested. Results show that CUDA's texture objects speeds-up 2D applications like image processing. We can achieve 99.26% speed-up with GPU kernels compared to the fastest CPU implementation. Also we can see a speed-up from 3.08 to 52.82 percent increase on global memory access when using texture memory. The project code is available on GitHub [1].**

***Index Terms*—image resizing, resampling, interpolation, lanczos, cuda**

I. Motivation and Significance

Image scaling is a fundamental operation in a wide range of applications, from medical imaging and satellite imagery to computer graphics, machine learning, and everyday photo editing. Poor interpolation can introduce artifacts such as blurring, blockiness, or ringing, which degrade visual quality and may affect downstream processing. High-quality interpolation methods like bicubic and Lanczos offer better visual fidelity but are computationally intensive, especially when applied to large images or in real-time systems. By implementing and optimizing these algorithms on GPU platforms, this project aims to accelerate the algorithms and preserve output quality, enabling scalable, high-performance image processing for both academic research and practical applications.

II. Problem Statement

Image resampling using interpolation methods yields new samples from the source image using sampling frequencies f_x in the horizontal direction and f_y in the vertical direction. The frequency is normalized such that if the sampling frequency is 1, it yields the same number of samples as the input image pixels; thus, it does not change the image size. The output image size can be found by Equation 1.

$$\begin{aligned}&(\text{OutWidth}, \text{OutHeight}, \text{Channel}) = \\ &(\lfloor f_x \cdot \text{InWidth} + 0.5 \rfloor, \lfloor f_y \cdot \text{InHeight} + 0.5 \rfloor, \text{Channel})\end{aligned} \tag{1}$$

The resampling methods discussed in this paper are channel-independent, meaning that all of the computations should be done per channel without interfering with each other.

A. Nearest Neighbor

Nearest neighbor interpolation assigns to each output pixel the value of the nearest pixel in the input image. This method is computationally efficient but introduces blockiness, especially during upscaling. We denote pixels of the input image as $f_{(x,y)}$. We can sample the image using nearest neighbor with Equation 2.

$$S(x,y) = f_{(\lfloor \frac{x}{f_x} + 0.5 \rfloor, \lfloor \frac{y}{f_y} + 0.5 \rfloor)} \tag{2}$$

B. Bilinear

Bilinear interpolation considers the closest 2x2 neighborhood of pixels and performs linear interpolation first in one axis and then the other. The method produces smoother results than nearest neighbor interpolation but may cause noticeable blurring. We can sample the image with this method using Equation 4.

$$p(t, f_0, f_1) = (1-t) \cdot f_0 + t \cdot f_1 \tag{3}$$

$$\begin{aligned} b_n &= p(x - \lfloor x \rfloor, f_{(\lfloor x \rfloor, \lfloor y \rfloor + n)}, f_{(\lfloor x \rfloor + 1, \lfloor y \rfloor + n)}) \\ S(x,y) &= p(y - \lfloor y \rfloor, b_0, b_1) \end{aligned} \tag{4}$$

C. Bicubic

Bicubic interpolation utilizes a 4x4 neighborhood and fits a cubic polynomial. This results in smoother and more visually appealing transitions, especially around edges and fine details. While computationally more expensive than bilinear interpolation, the visual quality is typically superior. For computational efficiency, the spline fitting and sampling are done by a kernel multiplication. We will use the kernel [2] to fit the Catmull-Rom spline [3], shown in Equation 5. For two-dimensional interpolation, the kernel is used to create four samples in one axis, then these samples are used in the other axis to obtain the final value, as formulated in Equation 6.

$$\begin{aligned} &p(t, f_{-1}, f_0, f_1, f_2) \\ &= \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\frac{1}{2} & 0 & \frac{1}{2} & 0 \\ 1 & -\frac{5}{2} & 2 & -\frac{1}{2} \\ -\frac{1}{2} & \frac{3}{2} & -\frac{3}{2} & \frac{1}{2} \end{bmatrix} \begin{bmatrix} f_{-1} \\ f_0 \\ f_1 \\ f_2 \end{bmatrix} \end{aligned} \tag{5}$$

$$b_n = p(x - \lfloor x \rfloor, f_{(\lfloor x \rfloor - 1, \lfloor y \rfloor + n)}, f_{(\lfloor x \rfloor, \lfloor y \rfloor + n)}, f_{(\lfloor x \rfloor + 1, \lfloor y \rfloor + n)}, f_{(\lfloor x \rfloor + 2, \lfloor y \rfloor + n)})$$

$$S(x, y) = p(y - \lfloor y \rfloor, b_{-1}, b_0, b_1, b_2) \quad (6)$$

D. Lanczos

Lanczos interpolation [4] is based on a windowed sinc function. The interpolation kernel is defined as in Equation 7.

$$L(x) = \begin{cases} \text{sinc}(x) \cdot \text{sinc}\left(\frac{x}{a}\right), & |x| < a \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

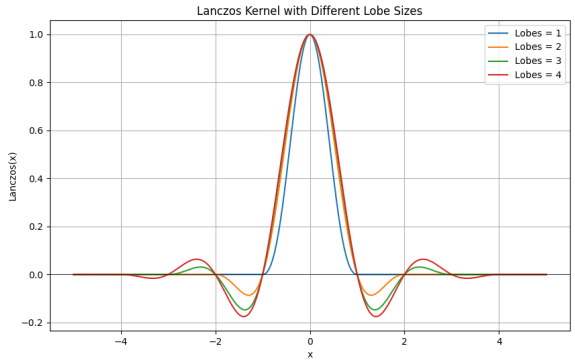

Fig. 1. Lanczos Kernel in 1 dimension

where a is the window size, and $\text{sinc}(x) = \frac{\sin(\pi x)}{\pi x}$. This method is known for sharp and high-quality results with minimal aliasing, though it may introduce ringing near discontinuities. We will use $a = 4$ as our window size, the same as the OpenCV library [5]. We acquire the two-dimensional sample by using Equation 8 on the x-axis, then using the results to sample on the y-axis, as formulated in Equation 9.

$$S(x) = \sum_{i=\lfloor x \rfloor - a + 1}^{\lfloor x \rfloor + a} f_i L(x - i) \quad (8)$$

$$S(x, y) = \sum_{i=\lfloor y \rfloor - a + 1}^{\lfloor y \rfloor + a} \sum_{j=\lfloor x \rfloor - a + 1}^{\lfloor x \rfloor + a} f_{(j,i)} L(x - j) L(y - i) \quad (9)$$

III. Prior Work and Limitations

OpenCV [5] is a rich computer vision library based on C++. It has a lot of functionality, including image resizing using the mentioned interpolation methods and their CUDA-accelerated versions, except for Lanczos on the GPU. We will use its image loading and saving functions for managing file input and output. We will also include their provided functions' performances in our comparisons. Multi-threading is disabled for CPU functions to get a true serial code execution reference.

IV. Algorithms

A command-line program is developed for each implementation approach. All of them read a specified image with OpenCV. The image is loaded to memory, where each pixel has three 8-bit values representing blue, green, and red color, respectively. Some algorithms may add an alpha channel to align the image data with a 32-bit addressable space. Programs also take f_x and f_y as inputs. The output image is saved to the same path as the input image, with its name extended with the program and algorithm name.

A. CPU

The 4 algorithms are implemented on the CPU by serially iterating over the y and x axes for the output image dimensions. Then the calculations explained in section II are implemented in C++. Two separate versions of implementations exist, one for arbitrary channels and the other for 4-channel images. This is done to test the effect of 32-bit pixel alignment on the performance.

B. CPU OpenCV

The 4 algorithms are already implemented on the CPU and provided by the OpenCV library. These will be tested and compared with the other implementations.

C. GPU

1) Output Sample Threads: The kernel is launched such that each kernel calculates one output sample. This approach simplifies writing the kernel but results in redundant global memory access, since multiple samples may require the same input pixels from the memory, depending on the sampling frequency. Shared memory is utilized to reduce this negative effect. A tile of corresponding input pixels is loaded from global memory to shared memory by each block of threads. Then, each thread reads the needed pixels from the shared memory.

2) Input Sample Threads: The kernel is launched such that each kernel loads one set of input pixels from the global memory and calculates the corresponding output samples. This results in less overall global access, but results in a longer serial code for each thread. Bicubic and Lanczos interpolation methods require neighboring input pixels to be loaded from global memory redundantly, which can be reduced with shared memory using the strategy described in subsubsection IV-C1.

D. GPU with Texture Object

CUDA provides texture objects, a way to store 1, 2, and 3-dimensional texture data in global memory that has useful features like border handling, coordinate and value normalization, on-memory interpolation, and spatial caching. Instead of passing linear memory arrays to the developed kernels, we create texture objects from the arrays and pass them. We can configure these objects to fit our needs. Value normalization maps the 8-bit integer values to floating-point values from 0 to 1. Interpolation performs bilinear interpolation on the requested coordinate. Value normalization has to be enabled to use the interpolation feature. The resulting texture object configurations for each algorithm can be seen in Table II.

TABLE I
EXECUTION TIMES (MS)

Run Configuration			CPU			GPU CUDA				
Implementation			OpenCV	Serial Code		OpenCV	Thread per Output	Thread per Input	Thread per Output	Thread per Input
Memory Interface			cv::Mat	Byte Array (3 channels)	Pixel (32-bit aligned)	cv::Mat	Byte Array (3 channels)	Pixel (32-bit aligned)	Texture Object	
Input Size	f_x, f_y	Algorithm								
1280x800	2.0, 2.0	Nearest N.	23.3378	7.3177	8.4436	2.4348	0.1208	0.0694	0.0625	0.1179
		Bilinear	26.9835	31.0424	31.4524	2.4818	0.1575	0.0860	0.07090	N.A.
		Bicubic	37.1332	85.0074	82.0809	2.8536	0.4469	0.1602	0.2869	0.1308
		Lanczos	137.029	336.95	255.601	N.A.	3.4005	1.2901	0.9511	1.2078
	4.0, 4.0	Nearest N.	59.1421	30.1391	27.9636	2.5886	0.4353	0.2583	0.2631	0.2843
		Bilinear	65.3907	124.043	124.526	2.7055	0.5866	0.4166	0.2682	N.A.
		Bicubic	84.8377	335.771	333.993	4.2121	1.7083	0.4154	0.9594	0.4145
		Lanczos	467.119	1365.17	1040.44	N.A.	13.3831	4.1845	3.5389	3.8567
5484x3926	2.0, 2.0	Nearest N.	252.027	160.350	139.042	4.0701	2.9594	1.8224	1.7837	1.7932
		Bilinear	329.080	672.017	667.466	4.2219	3.5352	1.8253	1.8137	N.A.
		Bicubic	443.262	1781.41	1760.05	12.8365	9.1422	2.6087	4.9825	2.0931
		Lanczos	2874.96	7254.20	5633.58	N.A.	64.9091	25.9909	19.1406	24.5529
	4.0, 4.0	Nearest N.	902.757	646.293	544.729	7.7173	11.2777	6.3166	6.1969	6.2306
		Bilinear	1124.31	2680.04	2720.24	8.7272	13.7935	8.2541	6.2046	N.A.
		Bicubic	1734.39	7298.53	7201.08	39.5795	35.7451	8.2885	19.3847	8.2415
		Lanczos	11260.8	29114.6	22458.3	N.A.	243.753	86.7789	67.0167	70.2026

TABLE II
TEXTURE OBJECT CONFIGURATIONS.

Algorithm	Normalization	Interpolation
Nearest Neighbor	No	No
Bilinear	Yes	Yes
Bicubic	Yes	No
Lanczos	Yes	No

E. GPU OpenCV

The algorithms, excluding Lanczos, are already implemented on the GPU and provided by the OpenCV library. These will be tested and compared with the other implementations.

V. EXPERIMENTS

All of the mentioned algorithms are tested 10 times for each run configuration. To ensure a just comparison, all the optimizations are enabled on all compilers. Example images with different sizes are used for scalability assessment. Experiment results can be seen in Table I. All image outputs are also examined to be identical for the used algorithm.

For CPU implementations, we can observe that OpenCV's implementation is faster in every case except for the nearest neighbor. When calculating GPU speed-up, the best-performing CPU code will be used for each algorithm. OpenCV's GPU implementation provides a substantial speed-up compared to CPU implementations, but falls behind custom GPU kernels. This shows that there is an overhead in OpenCV's implementation. Custom GPU kernel results show that using texture objects speeds up memory access.

Average speed-ups are calculated by averaging all percentage speed-ups for each run configuration. The results can be seen in Table III. We observe that the optimum solution for GPU kernels is when we launch threads per input pixel and use texture memory. The exception is for bilinear interpolation, because texture memory already provides hardware bilinear interpolation that can be easily used by launching threads per output sample. We achieved a 99.26% speed-up with the best GPU kernel compared to the fastest CPU implementation. Also, we can see a speed-up from 3.08 to a 52.82 percent increase on global memory access when using texture memory. This speed-up varies with the use of global memory, and since launching threads per input pixel, the global memory is not accessed as much as when we are launching threads per output sample, the observed speed-up is less. This shows the caching behaviour of texture memory, speeding up redundant memory accesses.

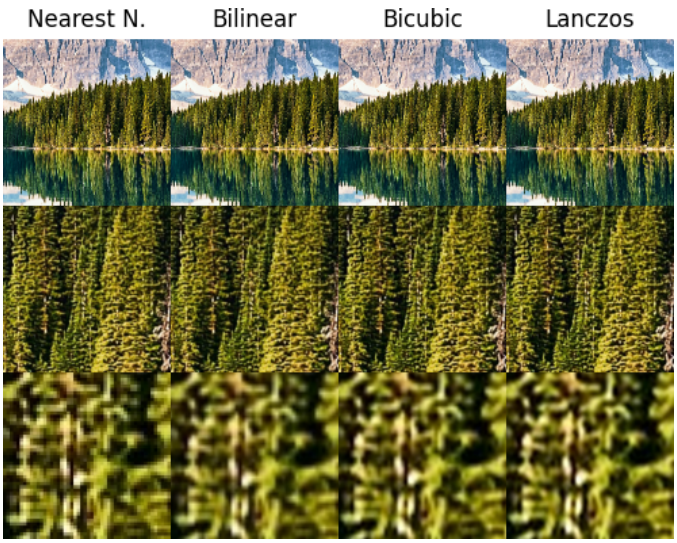

Fig. 2. Algorithm outputs.

TABLE III
AVERAGE PERCENTAGE SPEED-UPS

Reference \ Tested	CPU	GPU t/o	GPU t/i	GPU Tex. t/o	GPU Tex. t/i
CPU	0.0	98.23	99.24	99.2	99.26
GPU t/o	-6605.7	0.0	54.49	52.82	55.45
GPU t/i	-15131	-147.15	0.0	-15.84	3.08
GPU Tex. t/o	-14693	-131.57	-2.31	0.0	1.44
GPU Tex. t/i	-16350	-166.36	-7.55	-24.02	0.0

A. Environment

The test environment is shared for reproducibility. OpenCV is built from source to disable multi-threading and add the contributed CUDA package.

- Device: HP Victus 16 Laptop
- CPU: 13th Gen Intel Core™ i7-13700H 14 Core 20 Threads
- RAM: 64 GB
- GPU: NVIDIA GeForce RTX™ 4070 Laptop GPU 8GB VRAM
- Storage: 2TB Samsung NVMe SSD
- OS: Ubuntu 24.04.2 LTS
- CUDA Version: 12.9
- Compute Capability: 8.9
- CCCL Version: 2.8.4
- SM count: 36
- Max Threads per SM: 1536

VI. DISCUSSION AND FUTURE WORK

Comparing custom CPU code to OpenCV's implementation reveals many optimization techniques missing from our code. When we were implementing our GPU kernels, we also did calculations similar to the CPU code, aside from managing memory. This means there is a potential performance gain that is not explored. However, this matter didn't affect the main goal of the project, which was to explore and compare GPU memory interfaces and computational capabilities. The task of image interpolation is easily parallelized with minimal synchronization needed amongst threads. Despite this fact, there were many ways to approach and optimize the problem. In the future, lookup tables can be explored for replacing expensive cubic and sinusoidal calculations and optimizing the algorithm.

REFERENCES

[1] T. Acar, "CUDA Interpolation Kernels," Jun. 2025. [Online]. Available: https://github.com/tanacar2002/image-interpolation-cuda

[2] C. C. Lalescu, "Two hierarchies of spline interpolations. practical algorithms for multivariate higher order splines," *CoRR*, vol. abs/0905.3564, 2009. [Online]. Available: http://arxiv.org/abs/0905.3564

[3] E. Catmull and R. Rom, "A class of local interpolating splines," in *Computer Aided Geometric Design*, R. E. BARNHILL and R. F. RIESENFELD, Eds. Academic Press, 1974, pp. 317–326. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9780120790500500205

[4] C. E. Duchon, "Lanczos filtering in one and two dimensions," *Journal of Applied Meteorology and Climatology*, vol. 18, no. 8, pp. 1016 – 1022, 1979. [Online]. Available: https://journals.ametsoc.org/view/journals/apme/18/8/1520-0450_1979_018_1016_lfioat_2_0_co_2.xml

[5] G. Bradski, "The OpenCV Library," *Dr. Dobb's Journal of Software Tools*, 2000.