Parallel Image Upscaling with Anime4K

Yuning Zhang (yuningzh)

Jiaqiang Ruan (jruan)

Summary

We implemented the Anime4K[1] algorithm for anime-style image upscaling on the GHC machines, exploiting the GPU parallelism and the CPU parallelism respectively, and compared the performance of the two systems. Compared to a sequential implementation, our GPU implementation achieves a 58.93x speedup and the CPU one achieves a 10.54x speedup.

Background

Anime4K is an image upscaling algorithm specialized for anime-style images. Anime-style images tend to have crisp edges and sharp transitions between distinct colors. If we simply apply generic image upscaling algorithms on anime-style images, there will be noticable blurriness. Anime4K is an image upscaling algorithm specially designed based on the characteristics of anime-style images. Its basic idea is to push the filling color away from the edges to keep the linework thin.

The input of Anime4K is an image with low resolution and the output is an image with higher resolution.

Input (part, 100% size)

=>

Output (part, 100% size)

Approach

Sequential Implementation (Baseline)

The Anime4K algorithm is divided into several steps. Decode, upscale, luminance, thin lines, luminance, gradient and refine. In the sequential implementation of this algorithm, all computation is run sequentially for each pixel. The outer loop iterates over the rows and the inner loop iterates over the columns.

Step	Description
decode	The 3 color channels (RGB) of the original image are converted into floats ranging from 0.0 to 1.0.
linear	The original image is enlarged to the target size using bilinear interpolation.
luminance1	The luminance image, a grayscale one-channel image, is calculated from the enlarged image using a weighted average of RGB channels.
thin_lines	The thinner line image is calculated by using the luminance image and the enlarged image. This step matches patterns in the luminance image and calculates new color for each pixel with corresponding formulas. This makes the edges in the image thinner.
luminance2	Same as luminance1, a new luminance image is calculated from the thinner line image.
gradient	The gradient image is calculated based on the new luminance image with the Sobel filter. Each pixel represents the norm of the gradient of the corresponding pixel in the luminance image.
refine	The thinner line image is refined into the final output using the gradient image. This step matches patterns in the gradient image and calculates final color for each pixel with corresponding formulas.

OpenMP Implementation

The OpenMP version is implemented based on the sequential implementation, we added OpenMP pragmas to the outer loop of each step, which divides the image into horizontal strips. These strips will be processed parallely by OpenMP threads. The OpenMP scheduling is carefully tuned to get the best speedup.

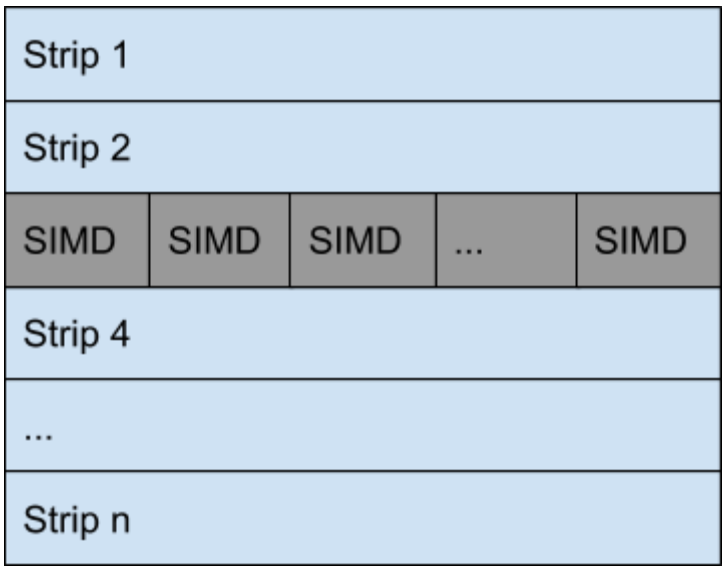
ISPC Implementation

The ISPC version is implemented based on the sequential implementation, we vectorized the inner loop of each step using SIMD instructions with ISPC.

To optimize the performance, we converted the data layout from array of structures to structure of arrays. In the sequential version, each image is represented by a WxH matrix where each pixel is represented by 3 floats (RGB), while in the ISPC version, the RGB channels are each represented by a WxH matrix, where the elements are float. This optimization reduces the unnecessary gather and scatter instructions. The speedup increases from 1.20x to 2.64x.

ISPC+OpenMP Implementation

To further improve the performance, we introduced the multi-core parallelism to the ISPC implementation. We utilized the task abstraction provided by ISPC. The ISPC compiler only generates calls to the runtime task system. In this project, we use a task system[3] backed by OpenMP.



In this Implementation, the image will be divided horizontally into strips. A task will be launched to handle each strip. The processing of each strip is further vectorized with SIMD instructions.

CUDA Implementation

There are two versions for the CUDA implementation. The global-memory version and the shared-memory version.

In the global-memory version, buffers for storing the enlarged image, luminance image, gradient image are declared in global memory, which is accessible from every thread block in the grid. To fulfill the convolution operation in the algorithm, two pixels are padded onto each side of the image. Therefore these temporary images have more pixels than the target output image. Then, the padded image is splitted into regions with maximum size of 32*32. Each region will be assigned to a thread block with 1024 threads. Therefore, each thread in the thread block will be

mapped to a certain pixel on the padded image. Of course, some threads might be mapped outside the padded image and for those threads we will ignore them. With this mapping relation, each step in the original algorithm is converted into a kernel function. Each kernel function specifies the calculation for each thread/pixel. The corner case of the calculation is checked and handled in each kernel function. And between kernel functions, block level synchronization is used to ensure the data dependency is fulfilled.

In the shared-memory version, buffers for storing the enlarged image, luminance image, gradient image are declared in block-level memory, which is only accessible within the same thread block in the grid rather than every thread block. Because of this restriction, these temporary images are first splitted into regions with the maximum size of 28*28 and then padded with 2 pixels on each side. The padded pixels are used to store the information of neighboring pixels outside the current region. The information is needed for calculating the result within the current region, since there is convolution operation in the algorithm. After this padding procedure, the padded region with the maximum size of 32*32 is assigned to a thread block with 1024 threads. This establishes a new mapping relation between pixel and thread. Then all steps of the original algorithm are implemented within one kernel function. Similarly, corner cases are checked and handled. Thread level synchronization is utilized. Note that, more idle and repeated computation is done in the shared-memory version compared to the global-memory version. Because there is padding for each region instead of the whole image.

Results

Performance Measurement

We use an application specific rate (frames per second) to measure the performance of our anime-style image upscaling implementation. If the code takes a wall-clock time of T second to upscale N images, the frames per second is defined to be N/T.

Experimental Setup

The benchmark images used in this project are selected from the benchmark image collection of the reference implementation[2]. The following table provides the description of the benchmark images.

Name	Size	Description
LS_Classroom	1920x1080	A simple scene with simple linework
KN_Shop	1920x1080	Simple foreground characters and complex background
MC_Paper	1920x1080	A simple scene with simple linework
GP_Crowd	1920x1080	A complex scene with complex linework
LS_Classroom720	1280x720	A 720p version of LS_Classroom

The benchmark framework benchmarks each implementation in the following way: it first loads a w x h sized image into the buffer, and then runs the selected implementation on the buffer for N times to upscale it into a W x H sized image. It measures the wall-clock time T of running the implementation. The wall-clock time measuring code is adapted from cycleTimer.h of Assignment 2. Finally, it calculates the frames per second (N/T).

Overall Performance

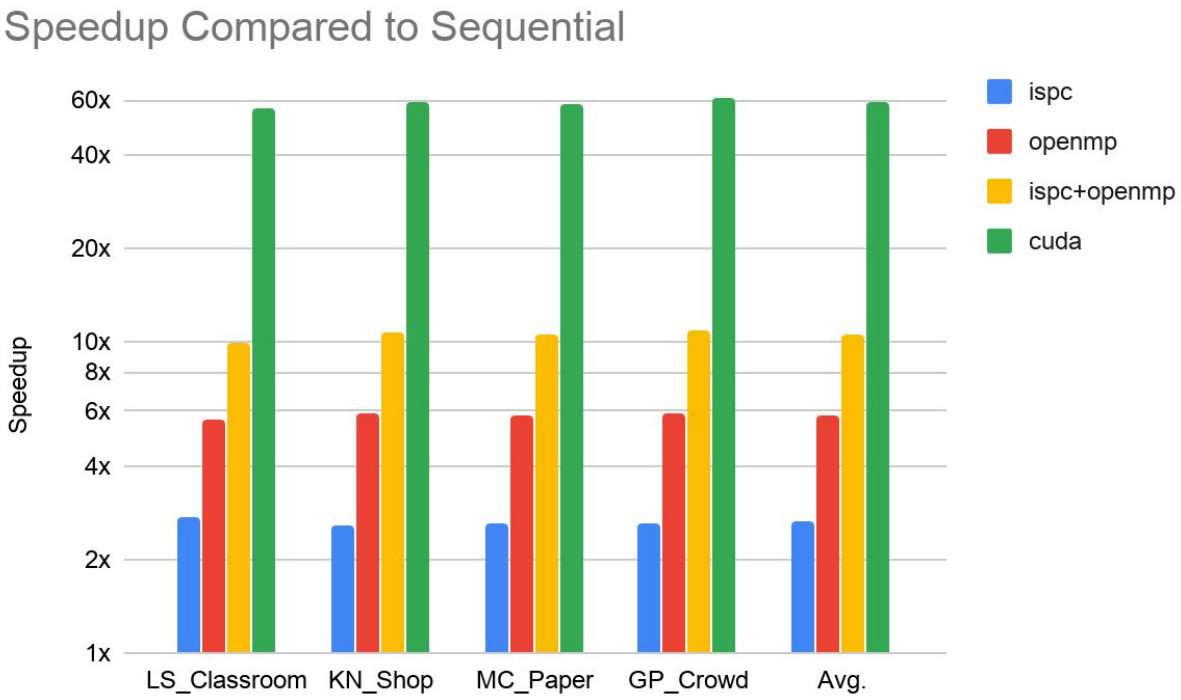
We have 5 versions of Anime4K implementation in our code:

1. sequential: without any SIMD and multi-threading. This will be used as the baseline.
2. ispc: ISPC version exploiting SIMD parallelism only.
3. openmp: OpenMP version.
4. ispc+openmp: ISPC version with a task system backed by OpenMP to exploit multi-threading.
5. cuda: CUDA version.

To measure the overall performance of our implementations, we benchmarked them by upscaling 1080p(1920x1080) images to 4k(3840x2160) images. (w=1920, h=1080, W=3840, H=2160) Most implementations are benchmarked with N=1000, except for sequential and ispc. These two implementations are benchmarked with N=100, due to the lower performance of these two implementations, otherwise it will take a very long time to finish.

	LS_Classroom	KN_Shop	MC_Paper	GP_Crowd	Avg. fps
sequential	2.76	2.52	2.55	2.45	2.57
ispc	7.60	6.51	6.62	6.37	6.78
openmp	15.53	14.81	14.89	14.35	14.90
ispc+openmp	27.58	27.01	26.88	26.90	27.09
cuda	156.98	149.95	149.58	149.28	151.45

The following graph shows the speedup of the optimized versions relative to the baseline. Note that the speedup axis is logarithmic.



The ISPC version with only SIMD yields a speedup around 2.64x. The OpenMP version yields a speedup around 5.80x. By combining ISPC and OpenMP, we get a speedup of around 10.54x, which is the best performance we can get on CPU. The CUDA version yields a speedup around 58.93x, which outperforms all the CPU implementations.

Influence of Image Content

The benchmark results show that the different image content will cause variations in the performance of the algorithm. Generally, the implementations work slightly better on simple scenes (LS_Classroom), and work slightly worse on complex scenes (GP_Crowd). We think the reason is that complex scenes have complex linework, which leads to higher divergence of computation in the algorithm.

However, the influence of image content on performance is small.

Scalability of Problem Size

To explore the scalability of problem size for our implementations, we benchmarked our code with different input image sizes (720p and 1080p), and different output sizes: 1080p(1920x1080), 2k(2560x1440), and 4k(3840x2160).

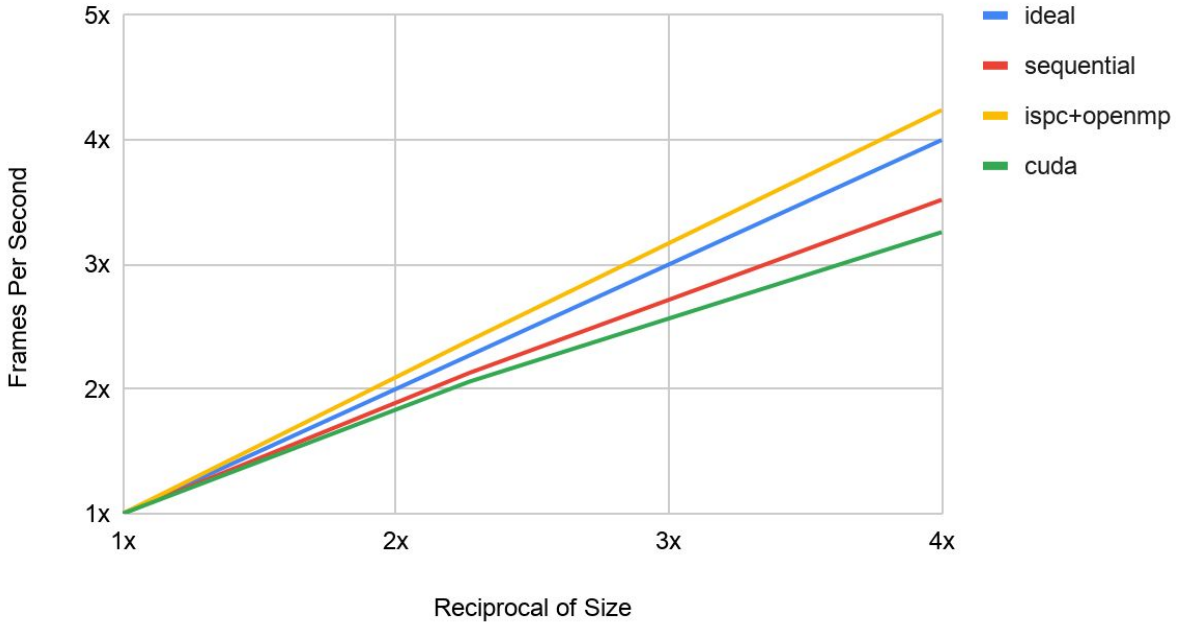
Image	W	H	Frames Per Second		
			sequential	ispc+openmp	cuda
LS_Classroom720	1920	1080	9.99	116.50	536.84
LS_Classroom720	2560	1440	6.05	65.65	339.37
LS_Classroom	2560	1440	5.59	65.44	302.22
LS_Classroom720	3840	2160	2.84	27.48	164.49
LS_Classroom	3840	2160	2.76	27.58	156.98

The result shows that the input image size only has a small influence on the performance. A smaller input size tends to have a slightly better performance. We think the reason is that a smaller input image leads to better cache locality. Also, for CUDA, a smaller input image means less data will be copied from host to device. The influence is small because all the steps iterate on WxH elements, which only depends on the output size.

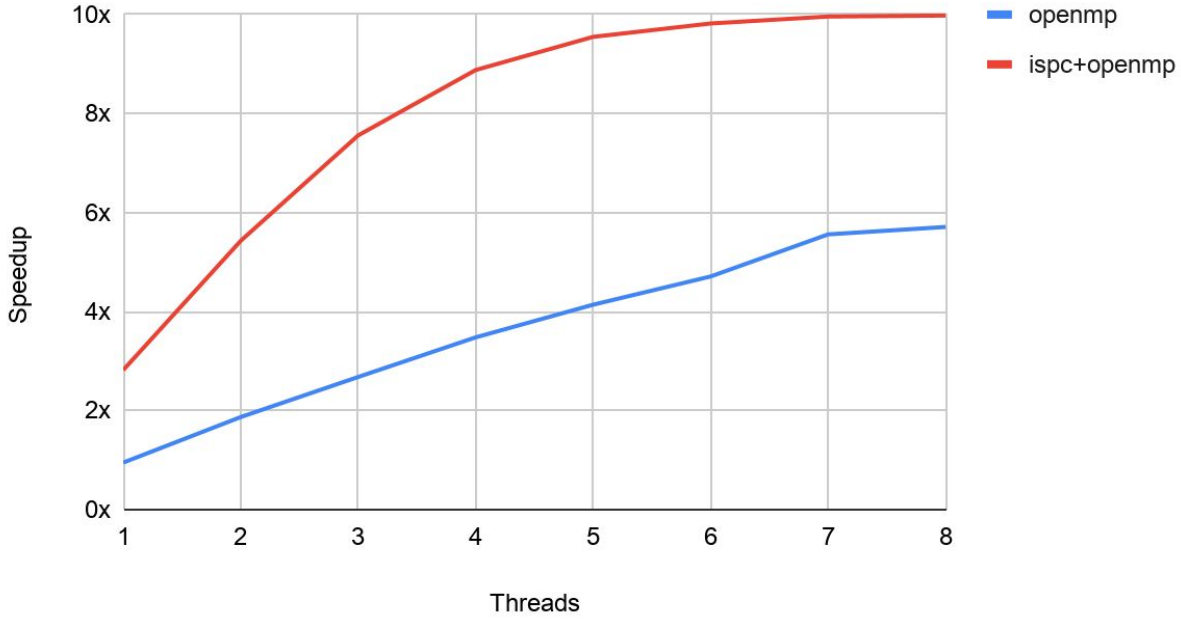
Output Size	Reciprocal of Size	sequential	ispc+openmp	cuda
1x	1x	1x	1x	1x
0.44x	2.27x	2.13x	2.39x	2.06x
0.25x	4x	3.52x	4.24x	3.26x

When the input size is fixed (720p), the performance of all the implementations increases proportionally to the reciprocal of the number of pixels in the output graph. This is because all the steps iterate on WxH elements. Therefore, the total amount of work is in proportion to the number of pixels in the output graph.

Performance vs Reciprocal of Output Size



Speedup vs Threads



Scalability of Threads

In general, this algorithm does not scale well with the number of threads. For the ISPC+OpenMP implementation, the benchmark results clearly show that there are diminishing marginal returns when adding more threads. For the OpenMP only implementation, when scale the number of threads from 1 to 7, the performance increases linearly. When increasing the number of threads from 7 to 8, the increment of performance slows down.

Our implementation divides the fixed amount of work and assigns them to threads. When increasing the number of threads, there will be less work for each thread. The threading overhead will take a larger proportion of running time. We guess that is the factor limiting the scalability with the number of threads.

Detailed Analysis of Multi-threading

To make a deeper analysis of the speedup we get from multi-threading, we instrumented our code to measure the execution time of each step.

Step	sequential time (ms)	openmp time (ms)	Speedup
decode	1999	266	7.52x
linear	9900	1303	7.60x
luminance	1883	1032	1.82x
thin_lines	9742	1397	6.97x
gradient	3327	519	6.41x
refine	9413	1432	6.57x
Elapsed	36266	5952	6.09x

The results show that in most of the steps, we get very good speedups, except for the compute luminance step. In the compute luminance step, we only get a 1.82x speedup. We guess this is because this step has a very low computational intensity: for each pixel, it needs to load 3 floats but only computes the weighted average of them. Therefore, in later implementations, we merged this step with other steps.

Detailed Analysis of SIMD

To make a deeper analysis of the speedup we get from SIMD, we instrumented our code to measure the execution time of each step. Note that in order to get better performance, in the ISPC implementation, we merged the compute luminance step with the linear upscaling step and the thin lines step.

Step	sequential time (ms)	ispc time (ms)	Speedup
decode	1998	200	9.99x
linear+lum	10836	2753	3.94x
thin_lines+lum	10664	5432	1.96x
gradient	3328	537	6.19x
refine	9394	4236	2.22x
Elapsed	36223	13161	2.75x

In some steps like decode and gradient, we get a very good speedup. In fact, in the decode step, we get a superlinear speedup (9.99x, while the number of SIMD lanes is only 8). We think the reason is that in the ISPC version, we used a trick to load the RGBA channels as a whole and then use bitwise operations to split the channels. However, this trick does not work in the sequential version.

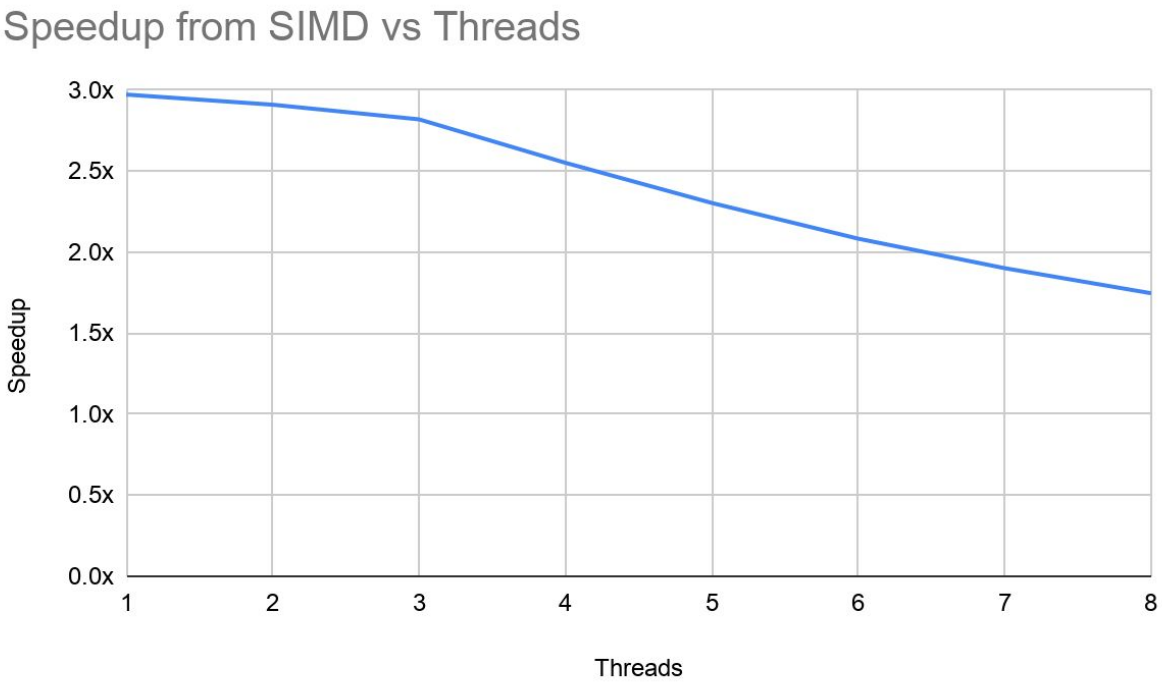
The linear upscaling step has a slightly worse speedup (3.94x). It is because this step requires gather operations (the original coordinate a pixel is mapped to is calculated dynamically). The gather operations are slower than normal load operations.

In steps like thin lines and refine, we get poor speedups (1.96x and 2.22x). We think the reason is that these steps have a high divergence nature (matching pixels with patterns and performing different operations). Therefore, the SIMD utilization is poor. We verified this hypothesis using the instrument feature provided by the ISPC compiler:

```
anime4k_kernel.ispc(0141) - if: expr mixed, true statements: 210576 calls, 40.68% active lanes
anime4k_kernel.ispc(0149) - if: expr mixed, true statements: 193215 calls, 40.17% active lanes
anime4k_kernel.ispc(0157) - if: expr mixed, true statements: 269207 calls, 31.86% active lanes
anime4k_kernel.ispc(0165) - if: expr mixed, true statements: 265704 calls, 30.72% active lanes
anime4k_kernel.ispc(0173) - if: expr mixed, true statements: 301025 calls, 29.93% active lanes
anime4k_kernel.ispc(0181) - if: expr mixed, true statements: 338724 calls, 26.56% active lanes
anime4k_kernel.ispc(0189) - if: expr mixed, true statements: 255693 calls, 30.70% active lanes
anime4k_kernel.ispc(0197) - if: expr mixed, true statements: 279035 calls, 29.61% active lanes
anime4k_kernel.ispc(0327) - if: expr mixed, true statements: 223663 calls, 38.50% active lanes
anime4k_kernel.ispc(0336) - if: expr mixed, true statements: 226417 calls, 38.32% active lanes
anime4k_kernel.ispc(0345) - if: expr mixed, true statements: 271275 calls, 20.97% active lanes
anime4k_kernel.ispc(0354) - if: expr mixed, true statements: 265779 calls, 21.15% active lanes
anime4k_kernel.ispc(0363) - if: expr mixed, true statements: 290573 calls, 21.59% active lanes
anime4k_kernel.ispc(0372) - if: expr mixed, true statements: 274729 calls, 22.74% active lanes
anime4k_kernel.ispc(0381) - if: expr mixed, true statements: 140149 calls, 15.08% active lanes
anime4k_kernel.ispc(0390) - if: expr mixed, true statements: 138142 calls, 15.07% active lanes
anime4k_kernel.ispc(0397) - if: expr mixed, true statements: 969101 calls, 60.23% active lanes
```

Interaction between SIMD and Multi-threading

To understand the interaction between SIMD and multi-threading, we calculate the speedup of the ispc+openmp implementation over the openmp only version with different number of threads.



The graph shows that as we increase the number of threads (increasing the multicore parallelism), we get less speedup from the SIMD parallelism.

To make a deeper analysis of this phenomenon, we divide the execution time into steps, and compare the speedup with the speedup from pure SIMD (ispc version over sequential version)

Step	openmp time (ms)	ispc+openmp time (ms)	Speedup	Pure SIMD Speedup
decode+linear+lum	2093	1041	2.01x	4.34x
thin_lines+lum	1903	1588	1.20x	1.96x
gradient	518	338	1.53x	6.19x
refine	1432	743	1.93x	2.22x
Elapsed	5947	3711	1.60x	2.75x

All steps get a smaller speedup from SIMD compared to using SIMD only. Therefore, there is a diminishing marginal return effect when combining multiple parallelisms. We think Amdahl's law can explain this phenomenon: the sequential part of the program will not get any speedup anyway. If we already make the parallelizable part fast by utilizing some parallelism, making it faster is less beneficial.

Also, we found that the CPU on the GHC machine is equipped with frequency scaling. When running with no multi-threading (with/without SIMD), it runs at 4.7GHz. When running with only multi-threading, it runs at 4.0GHz. When running with multi-threading and SIMD, it runs only at 3.8GHz. We think this hardware limitation also accounts for the speedup loss when combining parallelisms.

Detailed Analysis of CUDA

There are two versions of CUDA implementation, the global-memory version and the shared-memory version.

For the global-memory version, we analyze the speedup for each step of the algorithm.

Step	sequential time (ms)	cuda time (ms)	Speedup
copy between CPU GPU	0	398	
decode + linear	11899	56	212.48x
luminance	1883	67	28.10x
thin_lines	9742	95	102.55x
gradient	3327	25	133.08x
refine	9413	121	77.79x
Elapsed	36266	766	47.34x

As we can see, all steps have reasonable speedup.

1. The luminance step has lower speedup and we think the reason is that the computational intensity is low on this step. Luminance step requires three load operations. And these load operations are performed on the global memory on GPU, this requires large overhead. Therefore the speedup is not as high as other steps.
2. The overhead for copying data between CPU and GPU is very large, taking up 50% of the elapsed time.

In order to further explore the speedup capacity of GPU. We tried to implement the shared-memory version of cuda implementation. By using the shared-memory in the GPU instead of the global-memory, we hope it can reduce the communication time between the

memory and computation unit drastically. However, since different thread blocks cannot reach the data in the shared memory in other thread blocks. We need to pad the image region on each thread block to ensure the correctness of the algorithm. This introduces some repeated computation on those padded pixels on the edges of the regions.

	Sequential Implementation	Global-memory version	Shared-memory version
time for 100 frames (ms)	36266	766	637
fps	2.76	130.59	156.98
speedup	1.00x	47.34x	56.93x

Here is the analysis for the shared-memory version.

1. As we can see, using shared-memory instead of global memory can raise the speedup from 47.34x to 56.93x. Relative speedup is 1.20x.
2. If we assume the time spent on copying data between CPU and GPU is the same for both global-memory version and shared-memory version, we can calculate the relative speedup for other steps by changing from global-memory version to shared-memory version. (766-398)/(637-398) = 1.54x. Therefore, we can conclude that using shared-memory instead of global-memory has a large performance improvement. Even in our case, the shared-memory version has redundant computation compared to the global-memory version.

Discussion about Choice of Machine Target

The benchmark of our implementations shows that the GPU is a good fit for image processing workloads. Our CUDA implementation outperforms all the other CPU implementations. When implementing image processing applications, you should consider choosing GPU as the target first.

When the GPU is not available, we proved that you can still get a decent amount of speedup on the CPU by combining SIMD parallelism and multi-threading. The performance is enough for realtime(60fps) anime-style image upscaling when the output size is small(less than 2k, 2560x1440).

References

[1] B. Peng, 'Anime4K', 2019. [Online]. Available: https://github.com/bloc97/Anime4K/blob/master/Preprint.md. [Accessed: 04- May- 2020]
[2] bloc97, 'Anime4K-Comparisons', 2020. [Online]. Available: https://github.com/bloc97/Anime4K-Comparisons. [Accessed: 04- May- 2020]

[3] ispc, ' ispc/tasksys.cpp', 2020. [Online]. Available: https://github.com/ispc/ispc/blob/master/examples/tasksys.cpp. [Accessed: 06- May- 2020]

Division of Work

Yuning Zhang implemented all the versions of the algorithm on the CPU (including sequential, OpenMP, ISPC, and ISPC+OpenMP). Jiaqiang Ruan implemented all the versions of the algorithm on the GPU. Each one drafted the report for their own implementation.

Appendix

Project Code Link

Our code is available at https://github.com/codeworm96/parallel-anime4k.

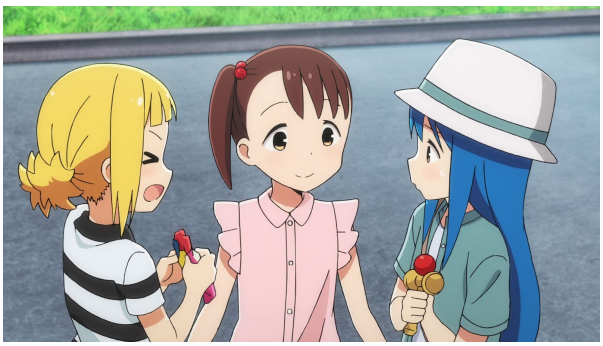
Benchmark Images

The benchmark images used in this project are selected from the benchmark image collection of the reference implementation[2]. The following table provides the description of the benchmark images.

Name	Size	Description
LS_Classroom	1920x1080	A simple scene with simple linework
KN_Shop	1920x1080	Simple foreground characters and complex background
MC_Paper	1920x1080	A simple scene with simple linework
GP_Crowd	1920x1080	A complex scene with complex linework
LS_Classroom720	1280x720	A 720p version of LS_Classroom

LS_Classroom

KN_Shop



MC_Paper

GP_Crowd