Natural Parallelism: Disclosing Efficient Techniques for Image Processing

Yohandi
School of Data Science
The Chinese University of Hong Kong, Shenzhen
YOHANDI@LINK.CUHK.EDU.CN

1 Introduction

1.1 Background

Natural parallelism often presents itself in computational problems where individual tasks do not rely on others. This inherent parallel nature is frequently seen in the domain of image processing, where actions on pixels are mostly independent. This project delves into this nature by utilizing multiple parallel programming paradigms to efficiently process images.

1.2 Objective

To expose different parallel programming languages and provide hands-on experience on how parallel programming operates in the context of image processing.

2 Parallel Programming: Models and Mechanisms

2.1 Understanding Types of Parallelism

There are three primary types of parallelism that computational systems usually exploit to achieve a faster and more efficient execution: Instruction Level Parallelism (ILP), Data Level Parallelism (DLP), and Thread Level Parallelism (TLP).

Instruction Level Parallelism (ILP) is where multiple instructions are executed concurrently, typically within a single processor. This kind of parallelism is inherent in the architecture of most modern CPUs. They leverage ILP through techniques such as pipelining, where different stages of multiple instructions are processed simultaneously; superscalar execution, where multiple execution units handle different instructions; and out-of-order execution, which reorders instruction sequences to minimize delays.

Data Level Parallelism (DLP) focuses on performing identical operations on multiple data points simultaneously. Instead of different instructions running concurrently, in DLP, the emphasis is on a single instruction being broadcast across multiple data streams. This type of parallelism is most prominent in Data Parallel Models. A classic example of DLP in action is the GPU (Graphics Processing Unit). GPUs are designed to execute the same instruction across many threads, where each thread handles different data. This makes GPUs especially suitable for tasks where a large dataset needs the same computational operation, such as graphics rendering or certain mathematical computations.

Lastly, Thread Level Parallelism (TLP) involves running multiple threads or processes in parallel. This form of parallelism can be seen in both shared memory architectures and distributed systems. In shared memory systems, such as multicore processors, multiple threads collaborate and operate concurrently on a single memory space. On the other hand, in distributed systems or computer clusters, separate processes operate on their own local memory and communicate with each other, often using protocols like MPI (Message Passing Interface). The essence of TLP is to break down computational tasks into smaller units (threads or processes) that can be executed in parallel, either on the same machine or across different machines.

Other types of parallelism, such as Request Level Parallelism (RLP) and Memory Level Parallelism (MLP), exist but won't be discussed in depth here.

2.2 Understanding Parallel Programming Models

In the realm of computational efficiency, six prominent parallel programming models are explored in this project, excluding the conventional sequential approach. Each model offers a unique methodology to exploit parallelism, tailored for specific tasks and hardware architectures.

SIMD (Single Instruction, Multiple Data) embodies the principle where a single instruction is dispatched to operate concurrently on multiple data points. This model is predominantly seen in vector processors and modern CPU architectures, emphasizing Data Level Parallelism (DLP). The sheer power of SIMD is manifested when the same operation, like an arithmetic or logical instruction, is required to be executed across large datasets.

MPI (Message Passing Interface) introduces a model of computation that thrives in distributed environments. Here, individual processes run independently, each with its local memory, and communicate by explicitly passing messages. This model excels in situations where tasks can be partitioned across different machines or cores in a cluster, making it adept for large-scale computations.

OpenMP offers a shared memory parallel programming paradigm. By utilizing directive-based annotations in codes written in languages like C, C++, or Fortran, OpenMP spawns multiple threads from a master thread, distributing tasks or iterations amongst them. Given its nature, it's particularly suited for multi-core processors where threads can operate on a shared memory space, facilitating easy data sharing and collaboration.

Pthreads (POSIX Threads) provides a more granular control over multi-threading compared to OpenMP. With a standardized set of thread creation and management APIs, developers manually delegate tasks among threads, allowing for more precise control. It's adaptable for any shared memory system, but with the caveat of additional management overhead.

CUDA (Compute Unified Device Architecture), an innovation by NVIDIA, allows developers to harness the computational power of GPUs for general-purpose processing. By defining functions, termed as kernels, that can be parallelly executed across a multitude of threads on GPUs, CUDA emphasizes both DLP and Thread Level Parallelism (TLP). This model is especially proficient for tasks that can be atomized into smaller independent chunks.

Lastly, OpenACC provides a directive-based approach to parallelism. By annotating sections of code, developers can indicate regions that can be offloaded to accelerators, such as GPUs. OpenACC abstracts many intricacies of accelerator programming, making parallel execution more accessible.

2.3 Similarities and Differences between Those Models

2.3.1 Similarities

- **Parallel Execution:** At their core, all models aim to increase computational speed and efficiency by executing multiple tasks or operations simultaneously.
- **Flexibility:** Most of these models, especially Pthreads, MPI, and CUDA, offer a high degree of flexibility, allowing for granular control over parallel tasks.
- **Interoperability:** These models can often be combined. For instance, one could use MPI for distributed parallelism and then employ OpenMP or CUDA within each MPI process for shared memory or data parallelism.
- **Platform Dependency:** Many of these models (especially CUDA and OpenACC) rely on specific hardware platforms or compilers. For example, CUDA is designed specifically for NVIDIA GPUs.

2.3.2 Differences

2.3.2.1 Operational Domain

- **SIMD:** Operates at the instruction level, executing the same instruction across multiple data elements.
- **MPI:** Built for distributed systems, where computation is split across multiple nodes or processors with separate memory spaces.
- **OpenMP and Pthreads:** Designed for shared memory systems, where multiple threads run on multi-core processors and access a common memory space.
- **CUDA:** Tailored for general-purpose computing on NVIDIA GPUs.
- **OpenACC:** Directive-based approach aiming at offloading computation to accelerators like GPUs.

2.3.2.2 Memory Model

- **Shared Memory:** OpenMP, Pthreads, CUDA, and OpenACC utilize a shared memory model where threads can directly access a common memory space.
- **Distributed Memory:** MPI follows a distributed memory model where each process has its own local memory, and data sharing is achieved through message passing.

2.3.2.3 Programming Complexity

- **Higher-level Abstraction:** OpenMP and OpenACC offer higher-level abstractions, using directives to indicate parallel regions.
- **Lower-level Control:** Pthreads, MPI, and CUDA provide more explicit control over parallel operations, which can offer greater flexibility but may increase complexity.

2.3.2.4 Scalability

- **Local Scalability:** OpenMP, Pthreads, and CUDA scale well on a single machine or GPU. Their performance is often bound by the number of cores or threads available locally.
- **Distributed Scalability:** MPI is designed for large-scale distributed computing, scaling across multiple nodes in a cluster or even across clusters.

2.3.2.5 Hardware Dependency

- **Platform-Specific:** CUDA is tied to NVIDIA GPUs, while OpenACC, although more versatile, is optimized for accelerators like GPUs.
- **Platform-Neutral:** MPI, OpenMP, and Pthreads are more neutral and can be implemented on a wide range of hardware platforms.

3 Refining Image Filtering: Parallel Processing and Memory Optimizations

In order to enchance the efficiency of the parallel program for Image Filtering, several optimization techniques were employed. One foundational approach was to circumvent the computational overhead of nested loops. Instead of relying on conventional iterative methods, parallel processing frameworks, such as OpenMP, were incorporated. For instance, by parallelizing the outer loop using OpenMP directives, like

```
#pragma omp parallel for
```

, the iterative computation could be distributed across multiple threads, significantly improving throughput.

Further, refinements in memory access patterns played an important role in optimization. Traditional methods, which often involve repeated access to the same address with additional calculations, were traded for more efficient strategies. One such method involved organizing image data into structures, specifically

```
std::vector<std::vector<Pixel>>
```

, where Pixel is a user-defined struct comprising r, g, b color values. This structure ensures a linear memory access pattern when looping through the image, promoting cache coherency and minimizing latency. Notably, the use of base pointers and pointer arithmetic, such as the employment of

```
reinterpret_cast<Pixel *>(input_jpeg.buffer)
```

, further streamlined memory access. This allowed for swift traversal of the image buffer, eliminating the occasional overhead associated with indexed access.

To further improve the parallelism, especially in the context of GPU processing with frameworks like CUDA, image partitioning techniques were employed. The image was divided into discrete blocks, with each GPU thread being assigned a pixel. This method of partitioning ensures balanced work distribution across threads, minimizing idle GPU cores and maximizing resource utilization.

For the sake of optimization, certain other refinements were also incorporated. For instance, datatype optimizations were made by replacing double-precision floating-point numbers with their single-precision counterparts, reducing memory bandwidth requirements without sacrificing significant precision.

Through the hybrid of these strategies, significant improvements in processing times and computational efficiency were observed, underscoring the power of parallel programming and the importance of thoughtful optimization.

4 Experiment Results

4.1 Part-A: RGB to Grayscale

Below is the table of execution time for various parallel programming models applied to the task of converting a 20K-RGB image to grayscale. Each programming model was tested on a different number of processes or cores, ranging from 1 to 32 (except for those that are not correlated with the number of processes or cores). The execution times are given in milliseconds (ms).

Processes / Cores	Sequential	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	639	429	639	716	593	27	28
2	N/A	N/A	789	644	594	N/A	N/A
4	N/A	N/A	503	340	451	N/A	N/A
8	N/A	N/A	351	178	278	N/A	N/A
16	N/A	N/A	284	97	185	N/A	N/A
32	N/A	N/A	242	73	136	N/A	N/A

Table 1: Comparison of different parallelization strategies in converting a 20K-RGB image to grayscale.

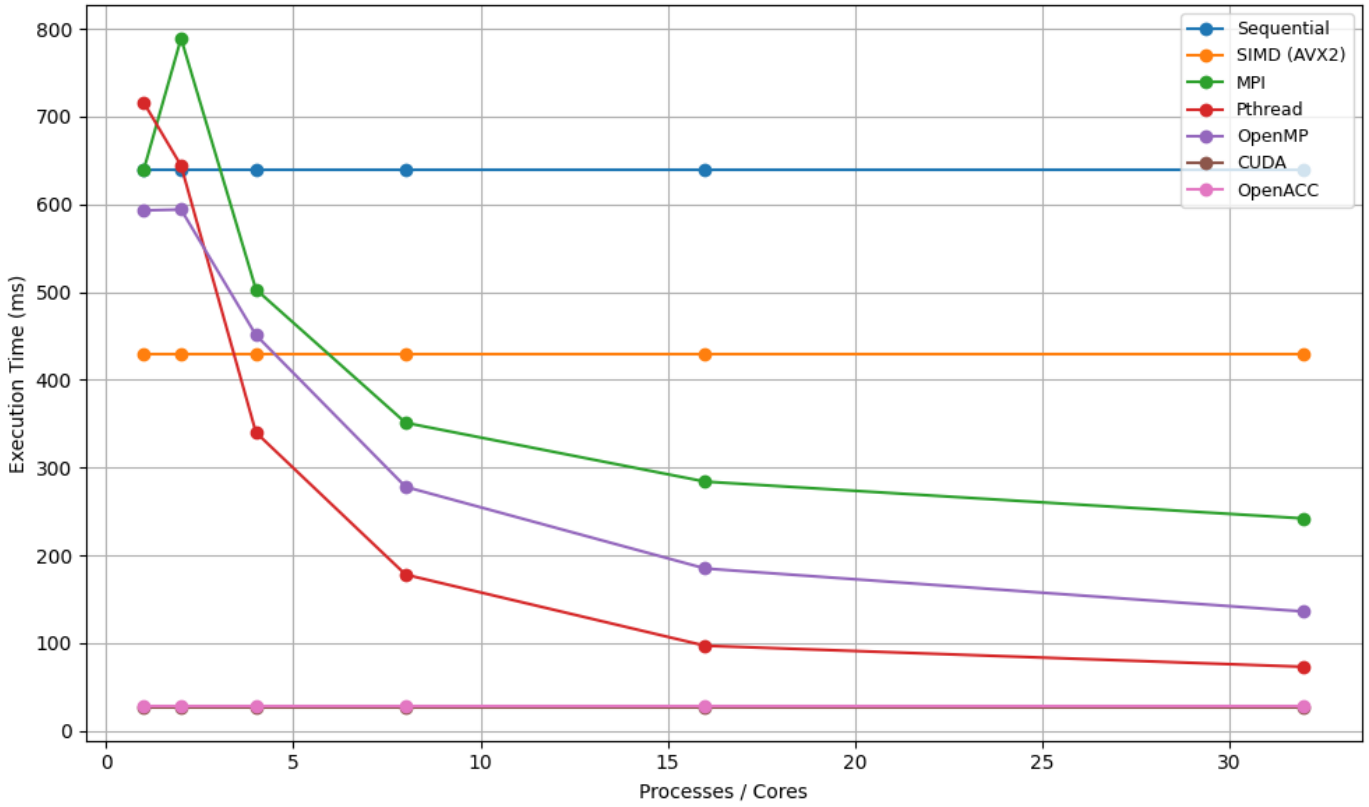

Figure 1: Streamline graph of different parallelization strategies in converting a 20K-RGB image to grayscale.

Using the single processor performance, both speedup and efficiency can be calculated as:

- $S(p) = \frac{t_s}{t_p}$, where t_s is the execution time using single processor performance and t_p is the execution time using multi-processor with p processors.
- $E = \frac{t_s}{t_p \times p} = \frac{S(p)}{p} \times 100\%$

The following tables represent the speedup and efficiency values for various parallelization strategies:

Processes / Cores	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	1.49	1.00	0.89	1.08	23.67	22.82
2	N/A	0.81	0.99	1.08	N/A	N/A
4	N/A	1.27	1.88	1.42	N/A	N/A
8	N/A	1.82	3.59	2.30	N/A	N/A
16	N/A	2.25	6.59	3.45	N/A	N/A
32	N/A	2.64	8.75	4.70	N/A	N/A

Table 2: Speedup factor for different parallelization strategies in converting a 20K-RGB image to grayscale.

Processes / Cores	SIMD (AVX2)	MPI	Pthread	OpenMP	CUDA	OpenACC
1	149%	100%	89%	108%	2367%	2282%
2	N/A	40.5%	49.5%	54%	N/A	N/A
4	N/A	31.75%	47%	35.5%	N/A	N/A
8	N/A	22.75%	44.88%	28.75%	N/A	N/A
16	N/A	14.06%	41.19%	21.56%	N/A	N/A
32	N/A	8.25%	27.34%	14.69%	N/A	N/A

Table 3: Efficiency percentage values for different parallelization strategies in converting a 20K-RGB image to grayscale.

4.2 Part-B: Image Filtering

Below is the table presenting a comparative analysis of different parallelization strategies applied to the task of applying a 3 x 3 filter to an RGB image. The table encompasses a variety of parallel programming models and their associated performances when executed on varying numbers of processes or cores. The performance metrics are characterized by both their average (AVG) and best (BEST) execution times in milliseconds (ms).

Model	**Processes / Cores**	**AVG**	**BEST**
Sequential	1	5149.8	5123
SIMD (AVX2)	1	1197.7	1194
MPI	1	7189.8	7169
	2	5969.9	5966
	4	3206.9	3202
	8	1784.2	1777
	16	1067.2	1064
	32	827.3	740
Pthread	1	4742.5	4724
	2	4188.5	4181
	4	2117.7	2115
	8	1088.6	1064
	16	550.2	545
	32	334	296
OpenMP	1	5901.2	5899
	2	5900.1	5897
	4	2988.8	2981
	8	1543	1494
	16	856.8	757
	32	461.7	394
CUDA	1	23.80652	23.7657
OpenACC	1	24	24
MPI + OpenMP	1 x 32	410	385
	2 x 16	623.2	531
	4 x 8	644.3	633
	8 x 4	717	707
	16 x 2	707.7	697
	32 x 1	816.8	752

Table 4: Comparison of different parallelization strategies with their average and best performances in filtering a 20K-RGB image.

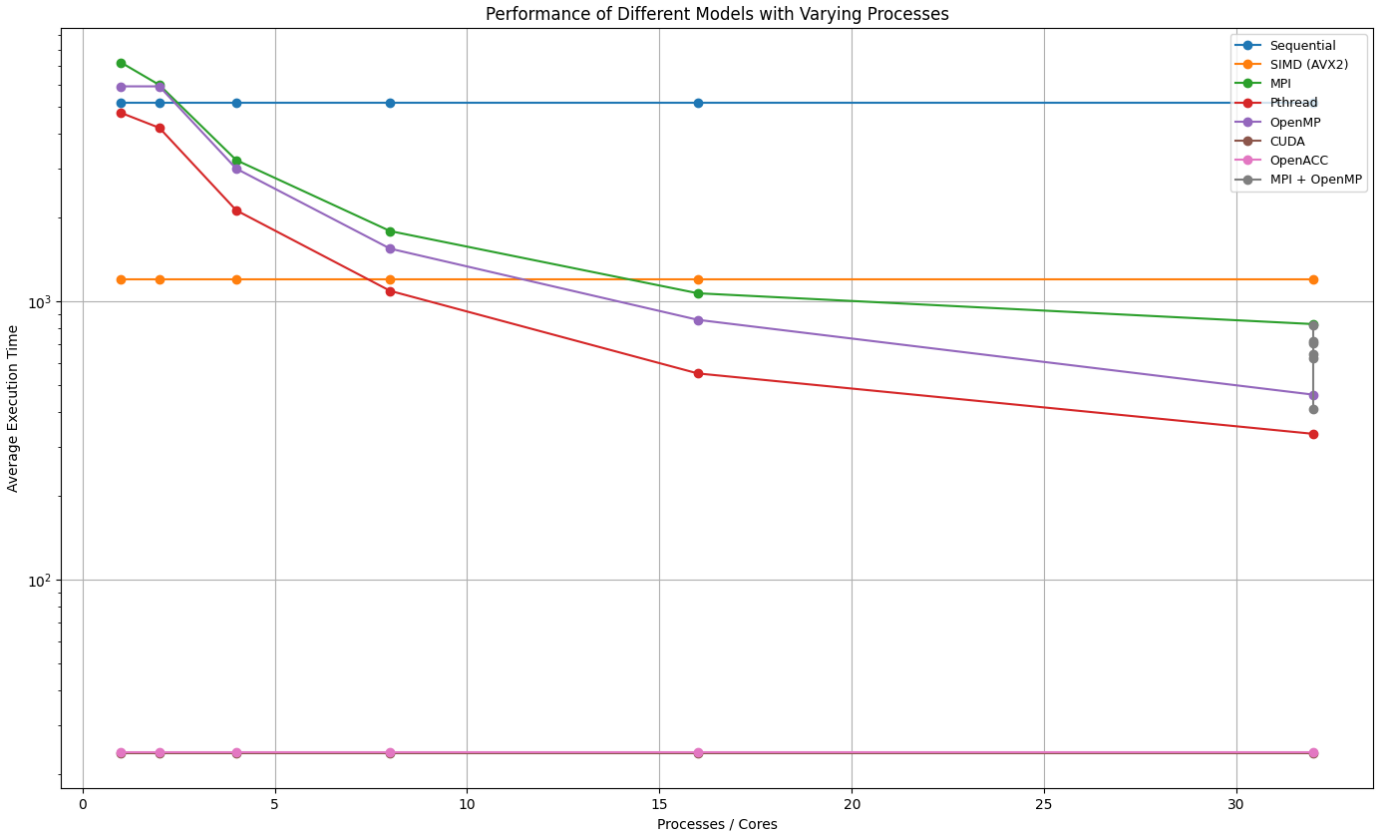

Figure 2: Streamline graph of different parallelization strategies in filtering a 20K-RGB image.

Using the single processor performance for the sequential model as the baseline, both the speedup and efficiency can be calculated accordingly using the previously mentioned formula. Based on the provided execution times, which is 4743.2 ms, the following tables represent the speedup and efficiency values for various parallelization strategies:

Model	Processes / Cores	Execution Time (ms)	Speedup $S(p)$	Efficiency E
Sequential	1	5149.8	1	100%
SIMD (AVX2)	1	1197.7	4.30	430%
MPI	1	7189.8	0.72	72%
	2	5969.9	0.86	43%
	4	3206.9	1.61	40.25%
	8	1784.2	2.89	36.13%
	16	1067.2	4.83	30.19%
	32	827.3	6.22	19.44%
Pthread	1	4742.5	1.09	109%
	2	4188.5	1.23	61.5%
	4	2117.7	2.43	60.75%
	8	1088.6	4.73	59.13%
	16	550.2	9.36	58.5%
	32	334	15.42	48.19%
OpenMP	1	5901.2	0.87	87%
	2	5900.1	0.87	43.5%
	4	2988.8	1.72	43%
	8	1543	3.34	41.75%
	16	856.8	6.01	37.57%
	32	461.7	11.15	34.85%
CUDA	1	23.80652	216.38	21638%
OpenACC	1	24	214.58	21458%

Table 5: Speedup factor and Efficiency percentage values for different parallelization strategies in filtering a 20K-RGB image.

5 Analyses and Findings

Upon examining the experimental results, insightful differences between the outcomes of PartA and PartB were observed. At the outset, PartA, which involved the RGB to Grayscale conversion, was relatively straightforward. The essence of its simplicity lay in the fact that each pixel's computation was independent of others, making it a prime candidate for embarrassingly parallel processing.

On the other hand, PartB, which centered on Image Filtering, presented a bit more intricacy. While it retained the inherent parallel nature, the computation for each pixel now depended on its neighboring pixels. This added layer of dependency meant that memory access patterns and data dependencies became more intertwined compared to PartA.

When comparing performance metrics, certain disparities emerged. Execution times in PartB might have shown variations owing to the added complexity of accessing neighboring pixel values. The intrinsic nature of operations in PartB, being more involved than a simple grayscale conversion in PartA, could have contributed to fluctuations in speedup or efficiency metrics. Additionally, the underlying memory access patterns – which grew more complex due to the neighboring pixel dependencies in PartB – may have had an influence on the cache behavior, potentially affecting the overall performance.

Having laid out these general observations, let's delve deeper into the specifics of each programming model's results. Through a closer examination of the data, we can better understand the nuances that led to the observed performance metrics and draw more concrete conclusions from our findings.

5.1 Part-A: RGB to Grayscale

5.1.1 CUDA's Exceptional Performance

CUDA exhibits an overwhelming speedup, especially for the single-core comparison. This massive speedup, surpassing 2000%, clearly demonstrates the parallel processing power of GPUs. Modern GPUs are designed with a high degree of parallelism, making them especially well-suited for tasks like image processing. The architecture of a GPU, with its numerous small cores, can handle many tasks concurrently, explaining the significant speedup.

5.1.2 Scalability in MPI

MPI shows increasing speedup as one increase the number of processors, suggesting good scalability. This is expected since the Message Passing Interface (MPI) is designed for distributed memory systems and can efficiently utilize multiple processors. By 32 cores, it exhibits a speedup of over 2.6, which is significant for CPU-based parallelism.

5.1.3 The Decline in Efficiency with Increased Cores

While the speedup increases with more cores, the efficiency for most methods decreases. This is a classic trade-off in parallel computing. As one add more processors, the overhead of managing these processors and communicating between them can reduce the overall efficiency. For instance, MPI's efficiency drops from 100% with one core to approximately 8.25% with 32 cores. Such a decline is typical in parallel computing, especially when nearing the hardware's core limits or when the task doesn't divide evenly among the cores.

5.1.4 Comparison of Pthreads and OpenMP

Pthreads and OpenMP also show increased speedup with more cores. Both are threading techniques for shared memory systems, and their performance profiles are somewhat similar. It's noteworthy that by 16 cores, Pthreads seems to have a better efficiency (41.19%) compared to OpenMP (21.56%). This might indicate that, for this particular problem, manual thread management (as in Pthreads) may offer better performance tuning than the more automated OpenMP approach.

5.1.5 SIMD (AVX2) versus Traditional Sequential

The SIMD (AVX2) approach offers a significant speedup even with a single core, indicating the benefits of vectorized operations on modern CPUs. SIMD operations can process multiple data points with a single instruction, which is why there's a notable performance improvement over the standard sequential method.

5.2 Part-B: Image Filtering

5.2.1 Superiority in GPU-Based Models: CUDA and OpenACC

The GPU-based models, specifically CUDA and OpenACC, recorded the shortest execution times at just 23.80652 ms and 24 ms, respectively. This phenomenal performance translates to speedups of 216.38 and 214.58 when compared to the sequential approach, showcasing the sheer power of GPUs in image-based operations. This prowess stems from their architecture designed for immense parallelism, making them perfect for tasks like 3x3 image filtering.

5.2.2 Scalability Insights in MPI

Starting with an execution time of 7189.8 ms for a single process, MPI showcased its scalability by reducing the time to 827.3 ms when using 32 processes. This significant decrease results in a speedup from 0.72 to 6.22 as the number of processes increased. This trend underscores the importance of using multiple processes for MPI to tap into its full potential, especially for tasks like image filtering.

5.2.3 Threaded Solutions: A Close Look at Pthreads and OpenMP

Both Pthreads and OpenMP displayed progressive improvement as the number of threads increased. Specifically, Pthreads achieved a top speedup of 15.42 with 32 threads, while OpenMP reached a speedup of 11.15 with the same. However, OpenMP's execution time showed diminishing returns past 8 threads, suggesting an optimal range for this model.

5.2.4 Efficiency Patterns Across Models

Efficiency, representing resource utilization, was exceptionally high for CUDA and OpenACC due to their low execution times. For other models, efficiency generally decreased with the addition of more processes or threads. For instance, MPI's efficiency was at its peak of 72% with one process but reduced to 19.44% with 32 processes. This numerical trend highlights the trade-off between speedup and resource utilization.

5.2.5 The Power of Hybridization: MPI + OpenMP versus Their Standalone Counterparts [Extra Credits]

The hybrid model combining MPI and OpenMP sought to amalgamate the distributed memory parallelism of MPI with the shared memory parallelism of OpenMP. This approach's objective was to leverage both models' strengths and potentially achieve better performance than when they are used individually.

Numerically speaking, the execution time for the hybrid approach was most optimal at a configuration of 1 MPI process combined with 32 OpenMP threads, registering a time of 410 ms. In contrast, standalone MPI with 32 processes clocked 827.3 ms and OpenMP with 32 threads had an execution time of 461.7 ms.

It's evident from the numbers that the hybrid approach with 1 MPI process and 32 OpenMP threads outperformed the standalone models, achieving a faster execution time by approximately 51.3 ms compared to standalone OpenMP and a staggering 417.3 ms compared to standalone MPI.

A plausible reason for this could be that with one MPI process, there is no inter-process communication overhead, and the entire computational might of the 32 OpenMP threads is dedicated to the task. In contrast, the standalone MPI model with 32 processes could introduce communication overheads, slowing down the execution, while the standalone OpenMP model doesn't harness the distributed memory capabilities offered by MPI.

In conclusion, for the 3x3 image filtering of a 20K-RGB image, the hybrid approach of MPI combined with OpenMP, particularly with the configuration of 1 MPI process and 32 OpenMP threads, provides a compelling performance advantage over the standalone MPI or OpenMP models.

5.2.6 The Promise of SIMD (AVX2): Embracing Eightfold Parallelism [Extra Credits]

One of the standout performers in the tests was the SIMD (AVX2) approach. Clocking in at 1197.7 ms, it boasts a speedup of 4.30 when pitted against the sequential model. SIMD, standing for Single

Instruction, Multiple Data, is an intrinsic aspect of parallel computing, and its benefits are distinctly visible in image processing endeavors such as this.

However, what's paramount to highlight here is the eightfold parallelism achieved by the AVX2 implementation. With its 256-bit wide vector registers, the AVX2 approach facilitates processing eight pixels in a single operation. This eight-at-once processing, contrasted against a scenario where each pixel would be processed sequentially, is a testament to how concurrently applying the same instruction to multiple data points can lead to profound performance enhancements.

The data clearly demonstrates that for tasks such as 3x3 image filtering, exploiting the parallelism offered by SIMD, especially when capable of processing eight data points simultaneously, isn't just an optimization; it's a transformative leap in computational efficiency. This underscores the importance and relevance of SIMD in modern high-performance computing, especially for data-intensive operations.

6 Conclusion and Overall Implications

Throughout the analysis, it's apparent that the realm of parallel programming is vast, encompassing diverse techniques and approaches. Each of these methodologies, from GPU-based models like CUDA and OpenACC to traditional multi-threading with Pthreads and OpenMP, possesses its unique strengths and challenges. One of the significant takeaways from PartA and PartB is the value of GPUs in parallel computation tasks, especially when dealing with large-scale image processing. The unparalleled speedup provided by GPU models underscores their potency in data-intensive operations.

The exploration of hybrid approaches, notably the combination of MPI and OpenMP, offers promising avenues for further optimization. By marrying distributed memory parallelism with shared memory parallelism, this hybrid approach excels in performance, besting standalone models. SIMD's (AVX2) capabilities, especially its eightfold parallelism, serves as another highlight, emphasizing the paradigm shift such parallelism brings to computational tasks. However, as efficiency across models indicated, there is an inherent trade-off between speedup and resource utilization, reminding us of the complexities in the world of parallel computing.

In summation, parallel programming, in all its varied forms, has transformative potential in accelerating computational tasks. Whether one leans towards GPU models, hybrid techniques, or SIMD optimizations, the essence lies in harnessing the power of concurrent processing. As technology continues to evolve, the pursuit of maximizing efficiency and speedup in parallel computing remains an ever-enticing challenge for researchers and practitioners alike.

Compilation and Execution Instructions

Within the submitted zip file, a folder project1 can be found, this will be our main project folder.

1. Navigate to the folder
2. Run run.sh or simply paste the below sequence of commands:

```
scancel -u {username} # to cancel submitted batch jobs
mkdir build
cd build
cmake ..
make -j4
cd ..
sbatch ./src/scripts/sbatch_PartA.sh
sbatch ./src/scripts/sbatch_PartB.sh
squeue # to check batch jobs queue
```

   By default, the above commands will execute the RGB to Grayscale transformation for all provided images and the Image Filtering only for the *20K* image. If one wish to test other images, one can simply modify the appropriate scripts located in src/scripts/. Additional scripts for different images might also be found in the parent directory, .. Make sure relevant images can be found in images/ directory.
3. Upon completion, one can find the experiment results in the directory: images/results/.

Appendix

After executing run.sh, the directory structure of images is as follows:

```
bash-4.2$ find . -print | sed -e 's;[^/]*/;|____;g;s;____|; |;g'
.
|____4k-RGB.jpg
|____4k-Smooth.jpg
|____Lena-RGB.jpg
|____Lena-Sharpen.jpg
|____Lena-Smooth.jpg
|____20K-RGB.jpg
|____results
| |____20K-Gray.jpg
| |____20K-smooth-sequential.jpg
| |____20K-smooth-simd.jpg
| |____20K-smooth-mpi-1.jpg
| |____20K-smooth-mpi-2.jpg
| |____20K-smooth-mpi-4.jpg
| |____20K-smooth-mpi-8.jpg
| |____20K-smooth-mpi-16.jpg
| |____20K-smooth-mpi-32.jpg
| |____20K-smooth-pthread-1.jpg
| |____20K-smooth-pthread-2.jpg
| |____20K-smooth-pthread-4.jpg
| |____20K-smooth-pthread-8.jpg
| |____20K-smooth-pthread-16.jpg
| |____20K-smooth-pthread-32.jpg
| |____20K-smooth-openmp-1.jpg
| |____20K-smooth-openmp-2.jpg
| |____20K-smooth-openmp-4.jpg
| |____20K-smooth-openmp-8.jpg
| |____20K-smooth-openmp-16.jpg
| |____20K-smooth-openmp-32.jpg
| |____20K-smooth-cuda.jpg
| |____20K-smooth-openacc.jpg
| |____20K-smooth-mpi-openmp-1x32.jpg
| |____20K-smooth-mpi-openmp-2x16.jpg
| |____20K-smooth-mpi-openmp-4x8.jpg
| |____20K-smooth-mpi-openmp-8x4.jpg
| |____20K-smooth-mpi-openmp-16x2.jpg
| |____20K-smooth-mpi-openmp-32x1.jpg
```

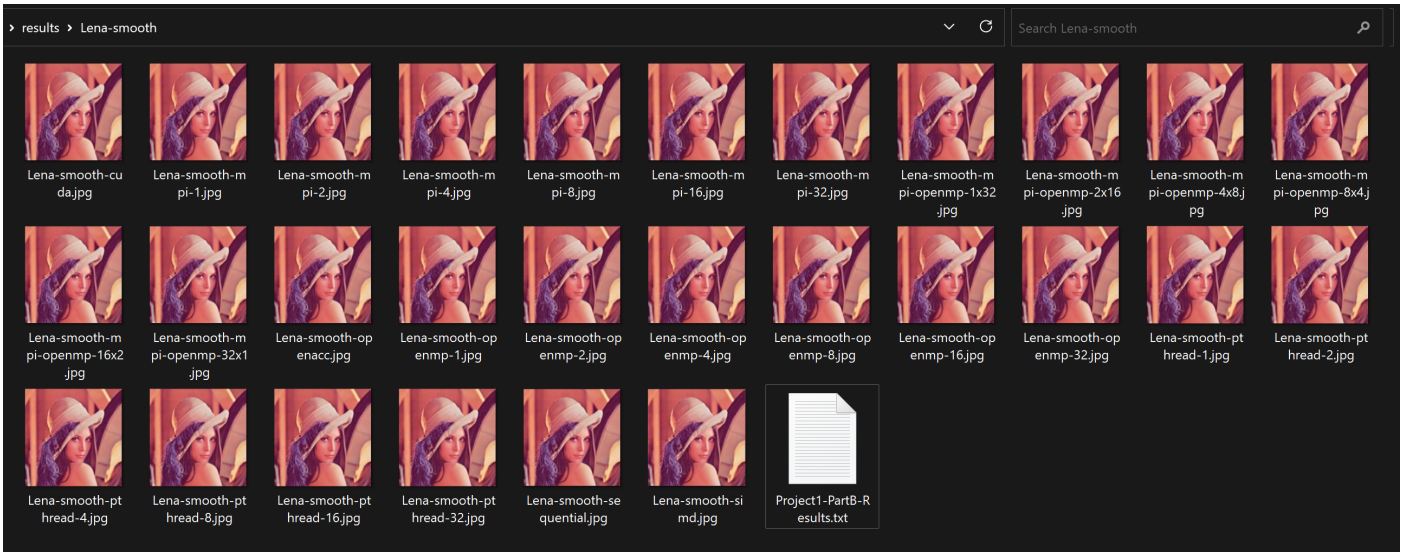

Figure 3: `../results` directory after filtering *Lena-RGB.png* using all models, including the hybrid one.

Model	Number of Processes / Cores	Iteration 1	Iteration 2	Iteration 3	Iteration 4	Iteration 5	Iteration 6	Iteration 7	Iteration 8	Iteration 9	Iteration 10	AVG	BEST
Sequential	1	5133	5123	5181	5130	5160	5147	5164	5171	5148	5141	5149.8	5123
SIMD (AVX2)	1	1198	1199	1199	1197	1197	1194	1199	1198	1198	1198	1197.7	1194
MPI	1	7177	7189	7202	7180	7203	7191	7201	7181	7205	7169	7189.8	7169
	2	5971	5968	5969	5973	5971	5969	5973	5969	5966	5970	5969.9	5966
	4	3211	3207	3206	3212	3208	3204	3209	3203	3202	3207	3206.9	3202
	8	1783	1780	1781	1780	1779	1777	1820	1780	1778	1784	1784.2	1777
	16	1070	1065	1068	1066	1067	1065	1070	1067	1064	1070	1067.2	1064
	32	914	753	742	749	740	823	942	813	857	940	827.3	740
Pthread	1	4746	4733	4739	4724	4735	4727	4748	4805	4728	4740	4742.5	4724
	2	4185	4181	4187	4185	4196	4185	4189	4182	4194	4201	4188.5	4181
	4	2121	2118	2117	2117	2119	2118	2116	2115	2117	2119	2117.7	2115
	8	1185	1153	1071	1074	1065	1070	1064	1071	1067	1066	1088.6	1064
	16	551	573	546	550	545	548	552	545	545	547	550.2	545
	32	355	370	360	352	360	298	296	298	319	332	334	296
OpenMP	1	5900	5902	5904	5900	5900	5903	5904	5899	5901	5899	5901.2	5899
	2	5900	5904	5901	5899	5897	5897	5905	5899	5898	5901	5900.1	5897
	4	2981	2985	2981	2987	2985	2990	3026	2984	2984	2985	2988.8	2981
	8	1494	1509	1671	1499	1513	1498	1543	1703	1495	1505	1543	1494
	16	900	904	908	758	757	767	918	945	950	761	856.8	757
	32	394	416	501	394	531	533	421	472	488	467	461.7	394
CUDA	1	23.7687	23.8266	23.8874	23.8403	23.8092	23.7794	23.7657	23.7907	23.8017	23.7955	23.80652	23.7657
OpenACC	1	24	24	24	24	24	24	24	24	24	24	24	24
MPI + OpenMP	1 x 32	466	390	437	388	389	424	385	428	406	387	410	385
	2 x 16	552	531	609	686	573	623	611	730	694	623	623.2	531
	4 x 8	639	663	634	636	666	633	666	633	633	640	644.3	633
	8 x 4	712	716	709	712	712	729	732	707	728	713	717	707
	16 x 2	702	714	714	717	701	713	701	697	697	721	707.7	697
	32 x 1	930	760	757	752	754	813	900	808	791	903	816.8	752

Figure 4: All models execution time comparison over 10 iterations for the 20K picture