Algorithmique parallèle et distribuée – Image Filtering

Yessin Moakher   Yuming Zhang

19 mars 2025

Introduction des GIFs

GIF File	Frames	Total Pixels	Load Time (s)
"fire.gif"	33	59400	0.001043
"9815573.gif"	10	1405000	0.018764
"TimelyHugeGnu.gif"	19	4372920	0.045146
"giphy-3.gif"	5	10366920	0.09594
"Mandelbrot-large.gif"	1	69682912	0.405383

Figure – Representative meta data for different GIF files. The dataset includes cases with many images but low pixel counts, single large images,etc.

Task Generation Strategy

The function `create_tasks_for_file` partitions GIF frames for distributing data.

- **Small GIF :** If total pixels are below `SMALL_FRAME_THRESHOLD`, they are processed as a single OpenMP thread in Rank0.
- **Large Frames :** If frame pixels exceed `LARGE_FRAME_THRESHOLD`, the frame is divided into row-wise blocks.
- **Multiple Frames :** If GIF has multiple frames, each frame is processed with these two criteria.

The number of blocks for a frame is computed as :

$$B = \frac{\text{frame\_pixels}}{\text{LARGE\_FRAME\_THRESHOLD}} + 1$$

MPI Task Distribution

A Master/Worker paradigm is used to distribute tasks dynamically :

- **Master Process (Rank 0)** :
  - Reads GIF metadata and generates tasks.
  - Assigns initial tasks and data to each worker.
  - Dynamically distributes tasks and data upon worker completion.
- **Worker Processes :**
  - Receive tasks and data via `MPI_IRecv`.
  - Process the assigned frame or frame block in parallel through OpenMP threads.
  - Send processed data and the corresponding task back to the Master and request new tasks.

Dynamic Scheduling of OpenMP Threads

Worker processes dynamically adjust the number of OpenMP threads :

- Extracts metadata from the received task to determine computational complexity.
- Dynamically sets the number of OpenMP threads based on task size.
- Balances workload within each worker to maximize efficiency.

Conclusion

- Task partitioning optimizes computational workload balance.
- MPI-based dynamic scheduling minimizes idle time across processes.
- Future improvements : decentralizing file I/O for better scalability.

openMP

- Use static scheduling for balanced workload distribution
- Optimize memory access by parallelizing the outer loop
- Use nowait() for independent loop sections
- Avoid parallelization for small loops where overhead is high

CUDA

Pipeline consists of :

- Transferring the input image to GPU memory
- Applying the Sobel filter
- Transferring the result back to host memory

CUDA Optimization

- Optimize memory access with shared memory
- Use halo regions for correct boundary handling
- Choose appropriate thread block sizes for better performance

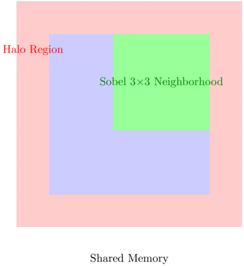

Figure 2: Illustration of the shared memory: the blue area represents the pixels corresponding to the block, the red area is the halo region, and the green area highlights a 3×3 neighborhood needed for the Sobel filter.

Conclusion

Each method has its advantages and disadvantages. **The final model will be a decision tree based on image size and available resources to determine the most suitable method.**

Conclusion

Each method has its advantages and disadvantages. **The final model will be a decision tree based on image size and available resources to determine the most suitable method.**