Arash Sal Moslehian

Canny Edge Detection

OpenMP and CUDA

OpenMP

Hardware:

Kernel: 5.15.29 x86_64

CPU: Intel i7-4710HQ (8) @ 3.500GHz

GPU: Intel 4th Gen Core Processor

GPU: NVIDIA GeForce GTX 850M

Memory: 12 GB

Disk: SATA SSD

Compiler: gcc (GCC) 10.3.0

Problem Size:

640x360
1280x720
1920x1080
2560x1440
3840x2160
7680x4320
15360x8640

Number of threads:

2 4 6 8 10 12 14 16

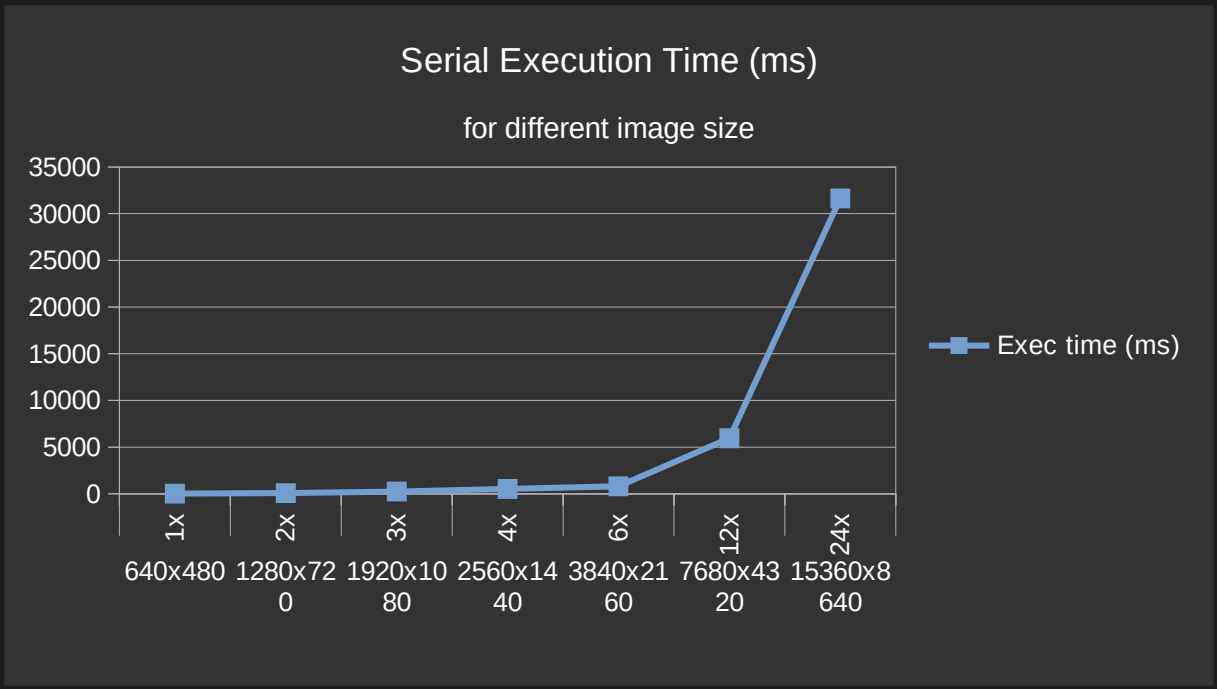
Serial Implementation

```
void gaussian_blur(const uint8_t *input_image, int height, int width,
                   uint8_t *output_image) {

  const double kernel[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
  const int kernel_sum = 16;

  for (int col = OFFSET; col < width - OFFSET; col++) {
    for (int row = OFFSET; row < height - OFFSET; row++) {
      double output_intensity = 0;
```

```
void gradient_magnitude_direction(const uint8_t *input_image, int height,
                                  int width, double *magnitude,
                                  uint8_t *direction) {
  const int8_t Gx[] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
  const int8_t Gy[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};

  for (int col = OFFSET; col < width - OFFSET; col++) {
    for (int row = OFFSET; row < height - OFFSET; row++) {
      double grad_x_sum = 0.0;
```

Serial Implementation (cont.)

Size	Size increase	Exec time (ms)	Exec time increase
640x480	1x	21.8	1
1280x720	2x	80.3	4
1920x1080	3x	244.6	11
2560x1440	4x	518.4	24
3840x2160	6x	806.2	37
7680x4320	12x	5941.1	273
15360x8640	24x	31632.7	1451



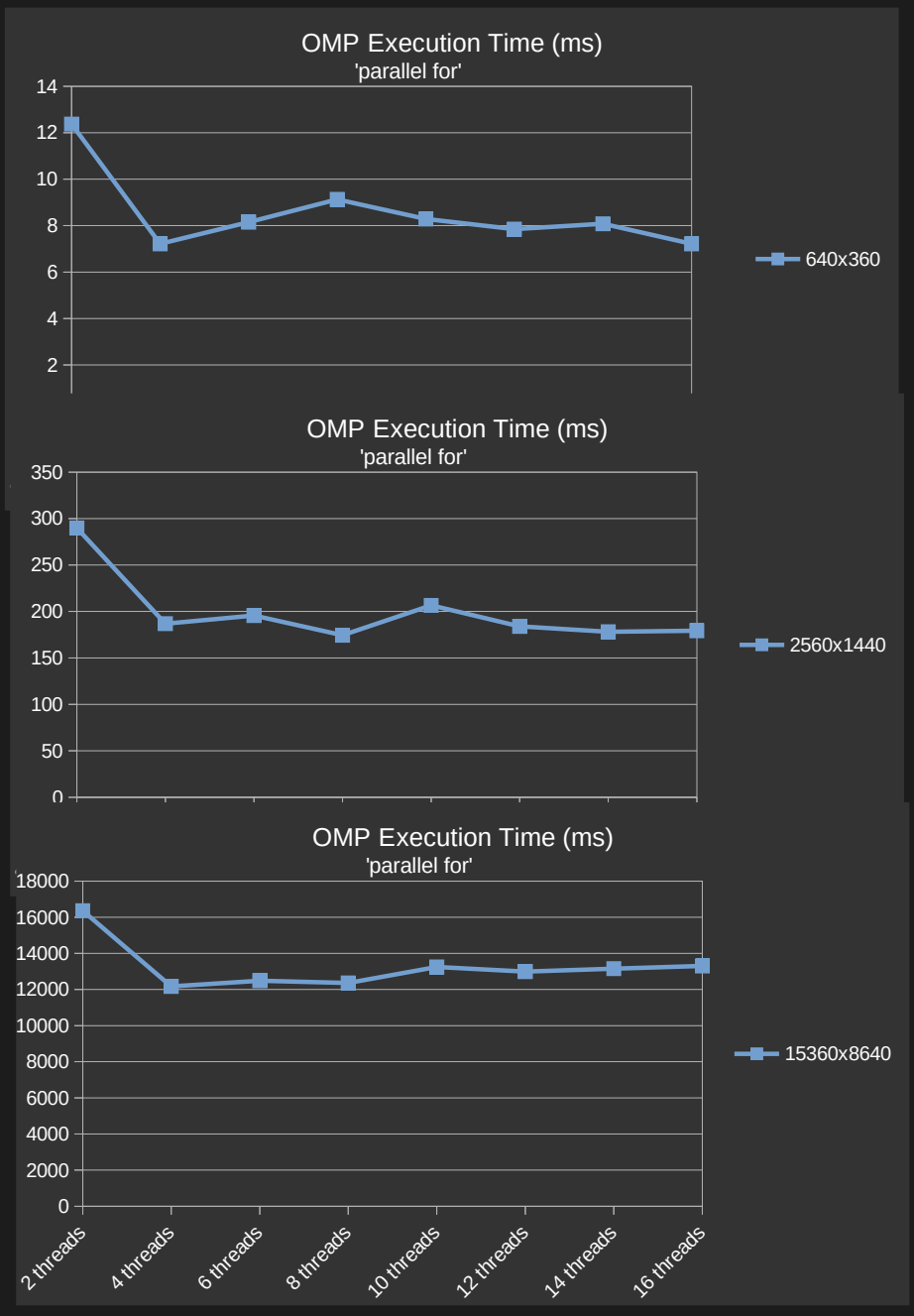
OMP Implementation
parallel for

```
void gaussian_blur(const uint8_t *input_image, int height, int width,
                   uint8_t *output_image) {

  const double kernel[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
  const int kernel_sum = 16;

#pragma omp parallel for
  for (int col = OFFSET; col < width - OFFSET; col++) {
    for (int row = OFFSET; row < height - OFFSET; row++) {
      double output_intensity = 0;
```

```
void gradient_magnitude_direction(const uint8_t *input_image, int height,
                                  int width, double *magnitude,
                                  uint8_t *direction) {
  const int8_t Gx[] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
  const int8_t Gy[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};

#pragma omp parallel for
  for (int col = OFFSET; col < width - OFFSET; col++) {
    for (int row = OFFSET; row < height - OFFSET; row++) {
      double grad_x_sum = 0.0;
```

OMP Implementation (cont.)
parallel for

Size	Size increase	Serial (ms)	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
640x360	1x	21.8	12.3728	7.216415	8.151925	9.13053	8.288115	7.848305	8.08104	7.21786
1280x720	2x	80.3	45.78355	29.07625	31.79085	27.3887	31.53275	30.3131	28.5131	31.1683
1920x1080	3x	244.6	135.3935	84.7534	82.45995	69.74105	83.32655	76.25445	74.2632	73.59955
2560x1440	4x	518.4	289.793	186.9385	195.7065	174.4685	206.5495	184.0395	178.1915	179.314
3840x2160	6x	806.2	468.617	334.09	396.8965	387.3095	406.6915	393.6035	381.81	393.6325
7680x4320	12x	5941.1	3437.8	2520.675	2530.315	2561.245	2813.315	2861.84	2855.24	2688.39
15360x8640	24x	31632.7	16359	12171.7	12485.6	12360.65	13242.95	12989.35	13155.8	13304.95



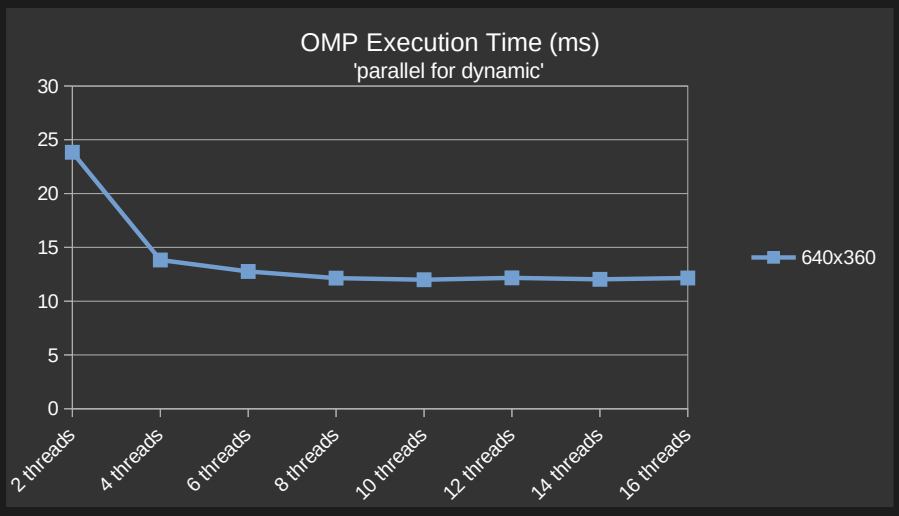
OMP Implementation
parallel for schedule(dynamic)

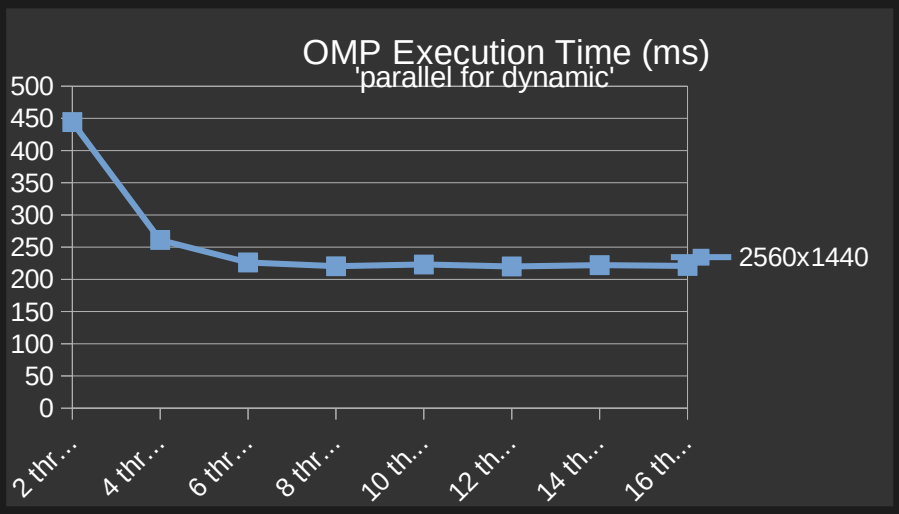
```
void gaussian_blur(const uint8_t *input_image, int height, int width,
                   uint8_t *output_image) {

 const double kernel[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
 const int kernel_sum = 16;

#pragma omp parallel for schedule(dynamic)
 for (int col = OFFSET; col < width - OFFSET; col++) {
   for (int row = OFFSET; row < height - OFFSET; row++) {
     double output_intensity = 0;
```

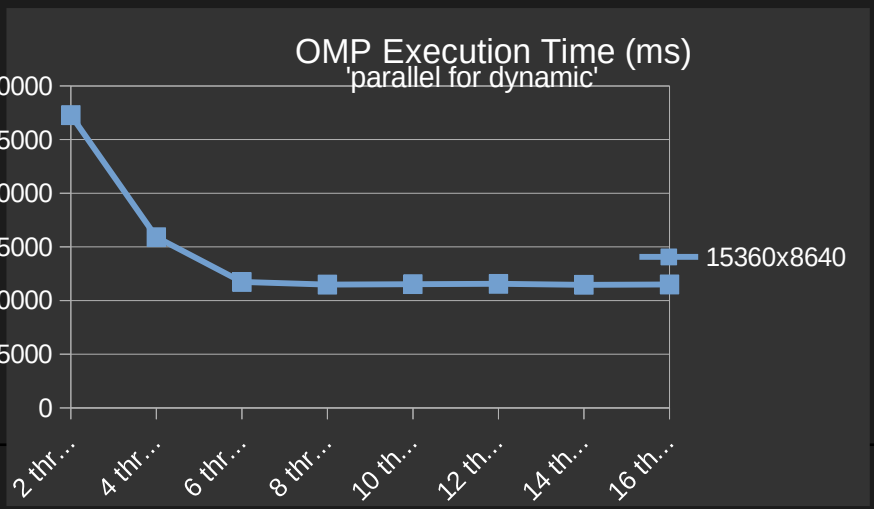
```
void gradient_magnitude_direction(const uint8_t *input_image, int height,
                                  int width, double *magnitude,
                                  uint8_t *direction) {
 const int8_t Gx[] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
 const int8_t Gy[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};

#pragma omp parallel for schedule(dynamic)
 for (int col = OFFSET; col < width - OFFSET; col++) {
   for (int row = OFFSET; row < height - OFFSET; row++) {
     double grad_x_sum = 0.0;
```

OMP Implementation (cont.)
parallel for schedule(dynamic)

Size	Size increase	Serial (ms)	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
640x360	1x	21.8	23.84465	13.8245	12.757	12.1454	11.987	12.1582	12.03225	12.1504
1280x720	2x	80.3	87.7179	50.65515	47.88115	45.9198	44.6141	45.21365	47.3189	45.239
1920x1080	3x	244.6	222.6115	127.557	116.8145	115.306	110.334	111.7655	108.922	110.232
2560x1440	4x	518.4	444.281	261.1385	226.215	220.394	223.089	220.011	222.176	220.7885
3840x2160	6x	806.2	788.5905	470.496	446.4375	432.7895	433.998	431.2925	439.952	438.8675
7680x4320	12x	5941.1	4510.25	2568.37	2328.39	2212.875	2207.43	2214.865	2223.965	2219.98
15360x8640	24x	31632.7	27274.45	15904.6	11733.6	11484.1	11528.1	11557.8	11464.6	11498.95







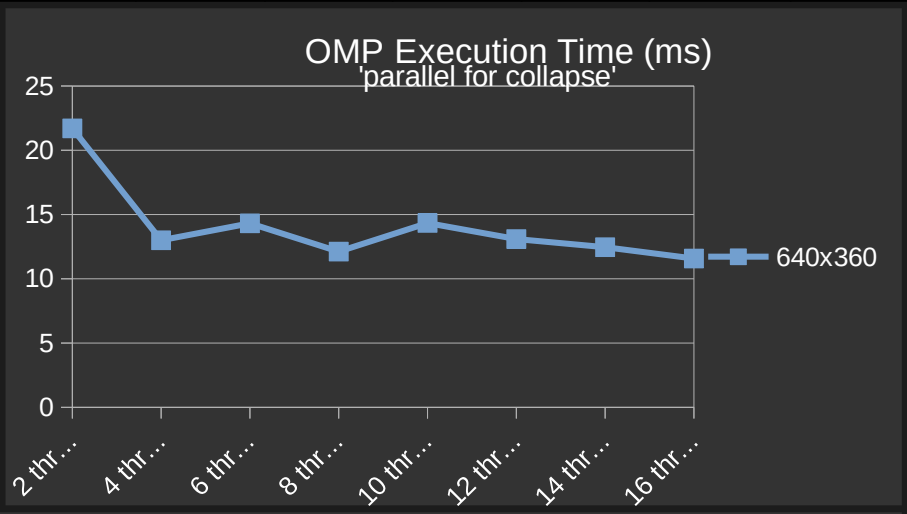
OMP Implementation
parallel for collapse(2)

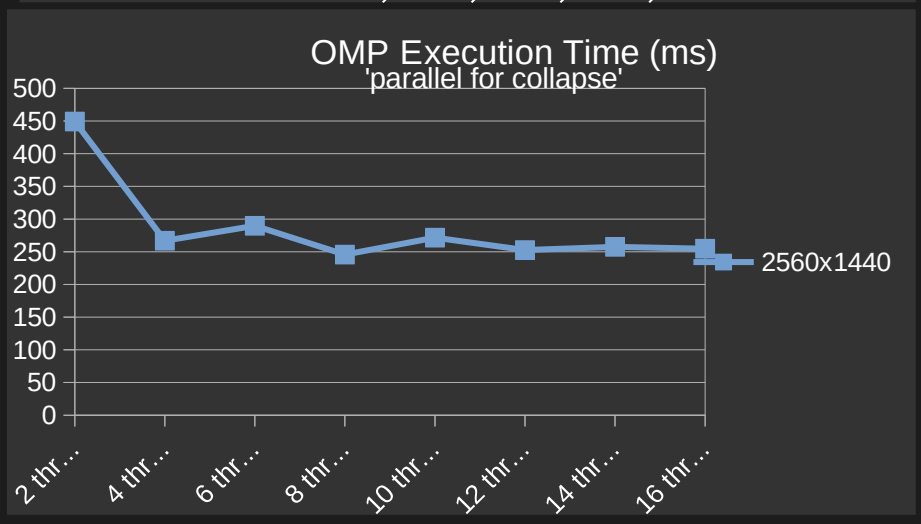
```
void gaussian_blur(const uint8_t *input_image, int height, int width,
                   uint8_t *output_image) {

  const double kernel[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
  const int kernel_sum = 16;

#pragma omp parallel for collapse(2)
  for (int col = OFFSET; col < width - OFFSET; col++) {
    for (int row = OFFSET; row < height - OFFSET; row++) {
```

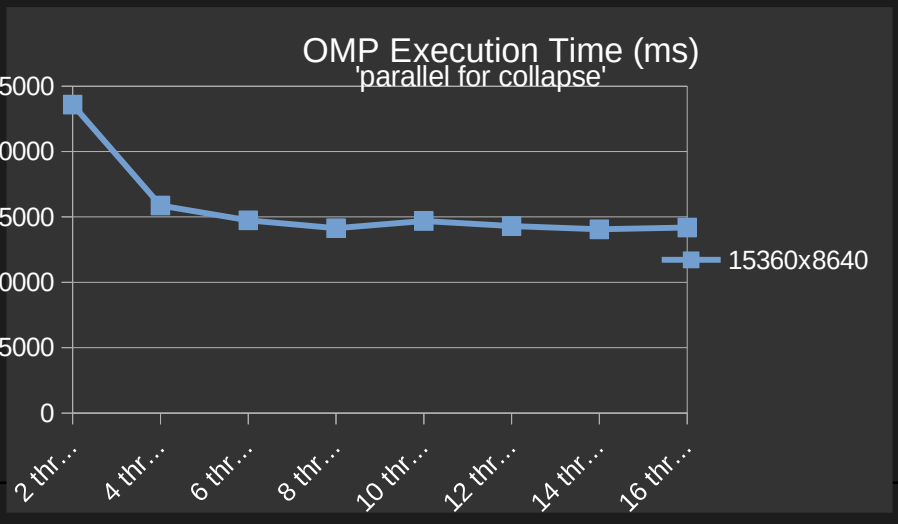
```
void gradient_magnitude_direction(const uint8_t *input_image, int height,
                                  int width, double *magnitude,
                                  uint8_t *direction) {
  const int8_t Gx[] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
  const int8_t Gy[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};

#pragma omp parallel for collapse(2)
  for (int col = OFFSET; col < width - OFFSET; col++) {
    for (int row = OFFSET; row < height - OFFSET; row++) {
      double grad_x_sum = 0.0;
```

OMP Implementation (cont.)
parallel for collapse(2)

Size	Size increase	Serial (ms)	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
640x360	1x	21.8	21.69815	12.9996	14.31765	12.1175	14.3357	13.092	12.4551	11.57515
1280x720	2x	80.3	79.9376	44.58995	55.1334	47.04535	55.1544	51.64735	47.6101	45.3153
1920x1080	3x	244.6	211.719	124.8545	136.164	110.156	125.724	119.5945	120.409	116.2535
2560x1440	4x	518.4	449.1775	266.8975	289.6915	245.679	271.481	252.682	257.413	254.7425
3840x2160	6x	806.2	779.2265	474.0115	526.4875	455.861	500.8465	481.636	473.5265	485.179
7680x4320	12x	5941.1	4635.69	2963.145	2948.88	2796.305	2994.05	2876.05	2942.24	2858.065
15360x8640	24x	31632.7	23587.65	15872	14742	14139.65	14691.35	14300.45	14054.95	14188.5







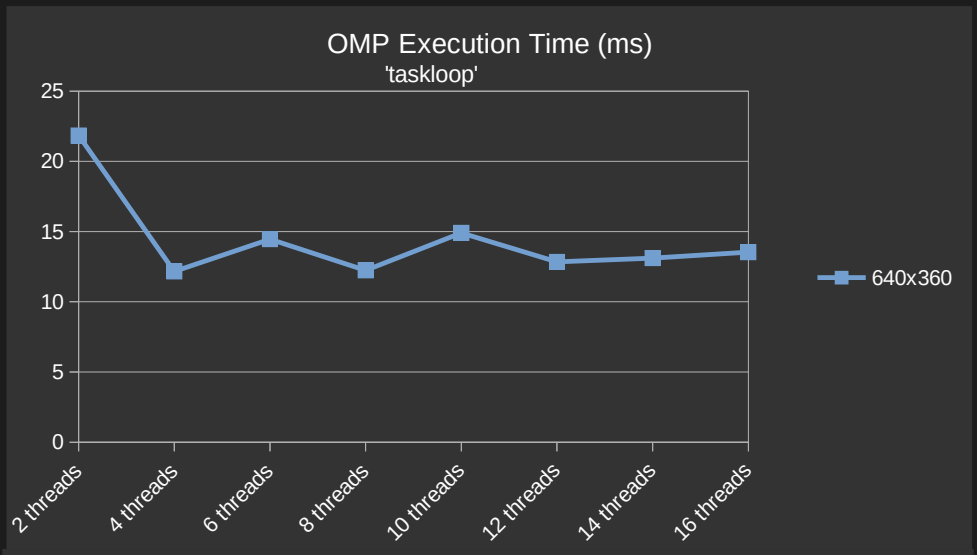
OMP Implementation
taskloop

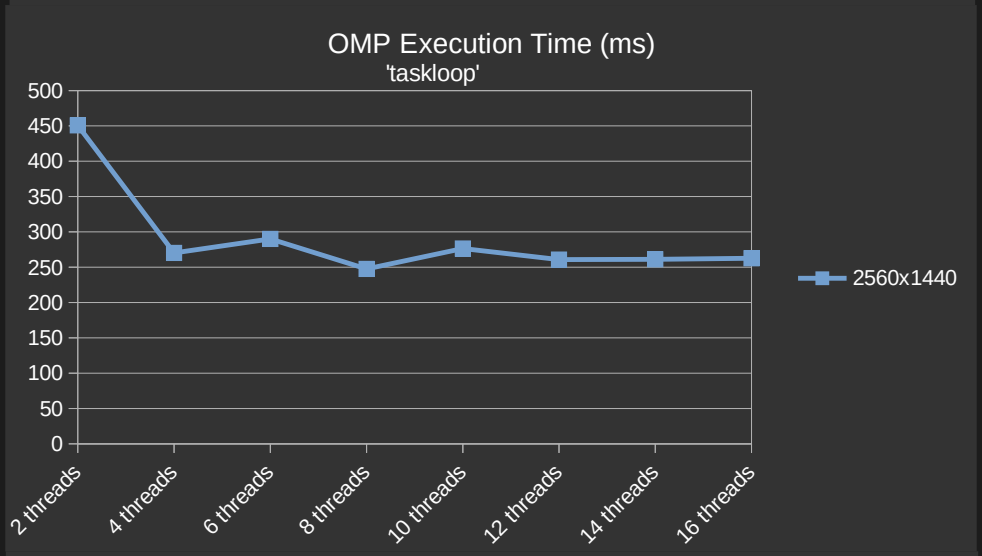
```
void gaussian_blur(const uint8_t *input_image, int height, int width,
                   uint8_t *output_image) {

  const double kernel[9] = {1, 2, 1, 2, 4, 2, 1, 2, 1};
  const int kernel_sum = 16;

  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp taskloop
      for (int col = OFFSET; col < width - OFFSET; col++) {
        for (int row = OFFSET; row < height - OFFSET; row++) {
          double output_intensity = 0;
```

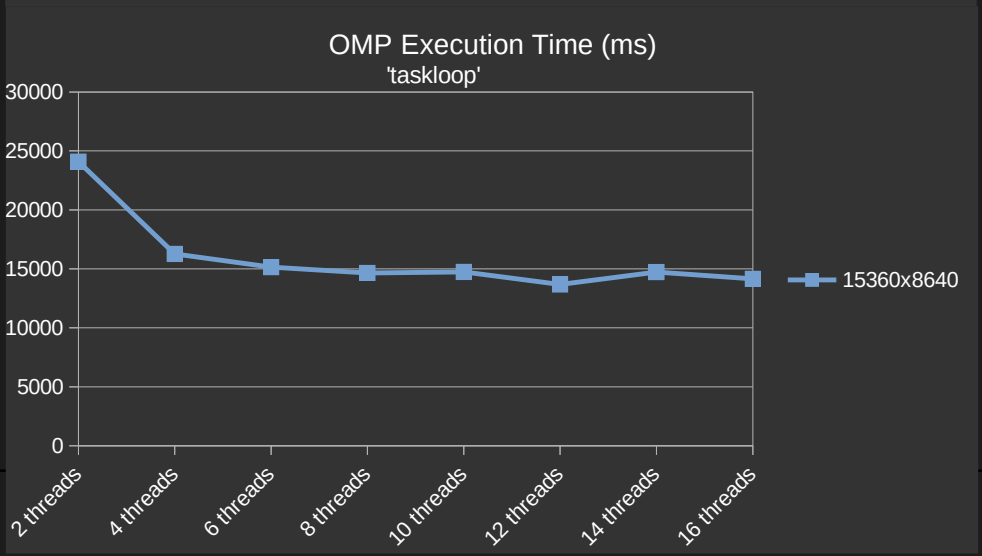
```
void gradient_magnitude_direction(const uint8_t *input_image, int height,
                                  int width, double *magnitude,
                                  uint8_t *direction) {
  const int8_t Gx[] = {-1, 0, 1, -2, 0, 2, -1, 0, 1};
  const int8_t Gy[] = {1, 2, 1, 0, 0, 0, -1, -2, -1};

  #pragma omp parallel
  {
    #pragma omp single
    {
      #pragma omp taskloop
      for (int col = OFFSET; col < width - OFFSET; col++) {
        for (int row = OFFSET; row < height - OFFSET; row++) {
          double grad_x_sum = 0.0;
```

OMP Implementation (cont.)
taskloop

Size	Size increase	Serial (ms)	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
640x360	1x	21.8	21.8384	12.1669	14.4651	12.23775	14.9045	12.843	13.11225	13.5336
1280x720	2x	80.3	78.3026	51.88225	54.1119	44.94625	54.291	49.0512	48.19575	46.9351
1920x1080	3x	244.6	212.7445	125.6585	134.078	110.1215	127.3895	119.4885	115.1575	115.519
2560x1440	4x	518.4	450.722	270.163	290.031	247.6295	276.4785	260.821	261.1705	262.6315
3840x2160	6x	806.2	778.49	461.7065	523.9365	460.31	512.002	492.4445	507.2325	489.8985
7680x4320	12x	5941.1	4600.935	2957.93	3019.495	2780.225	2978.2	2976.785	2933.965	2921.015
15360x8640	24x	31632.7	24091.5	16267.65	15148.55	14657	14741.7	13676.65	14731.8	14159.55



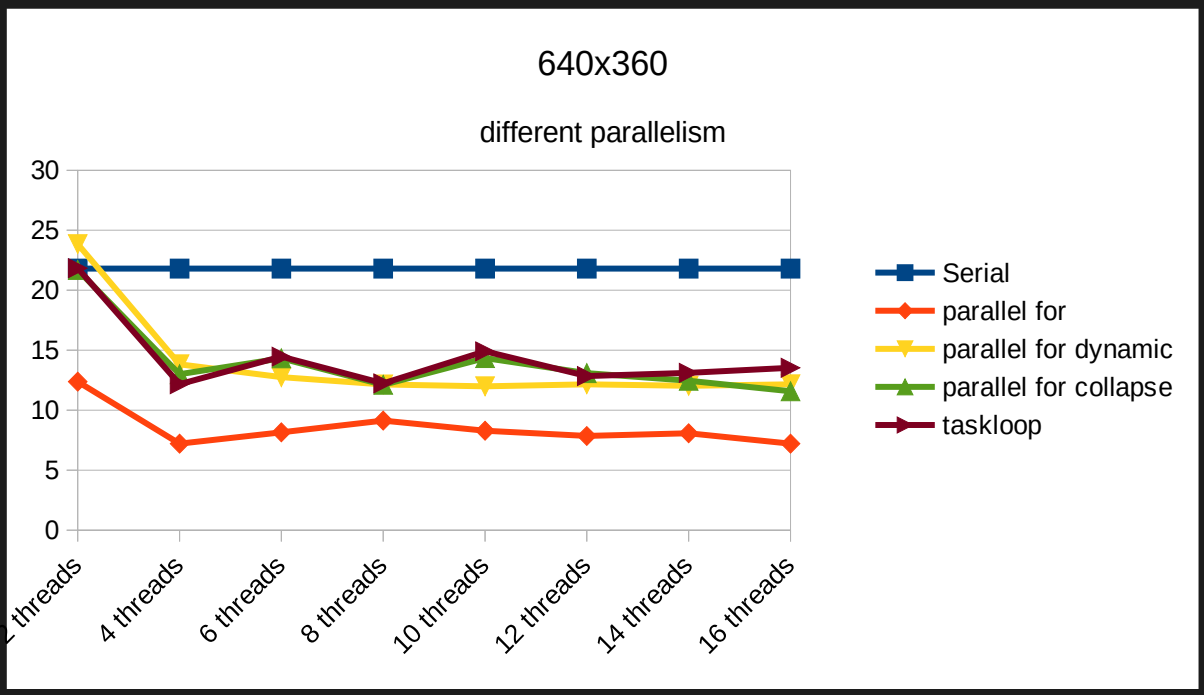




Parallelism Comparison

640x360

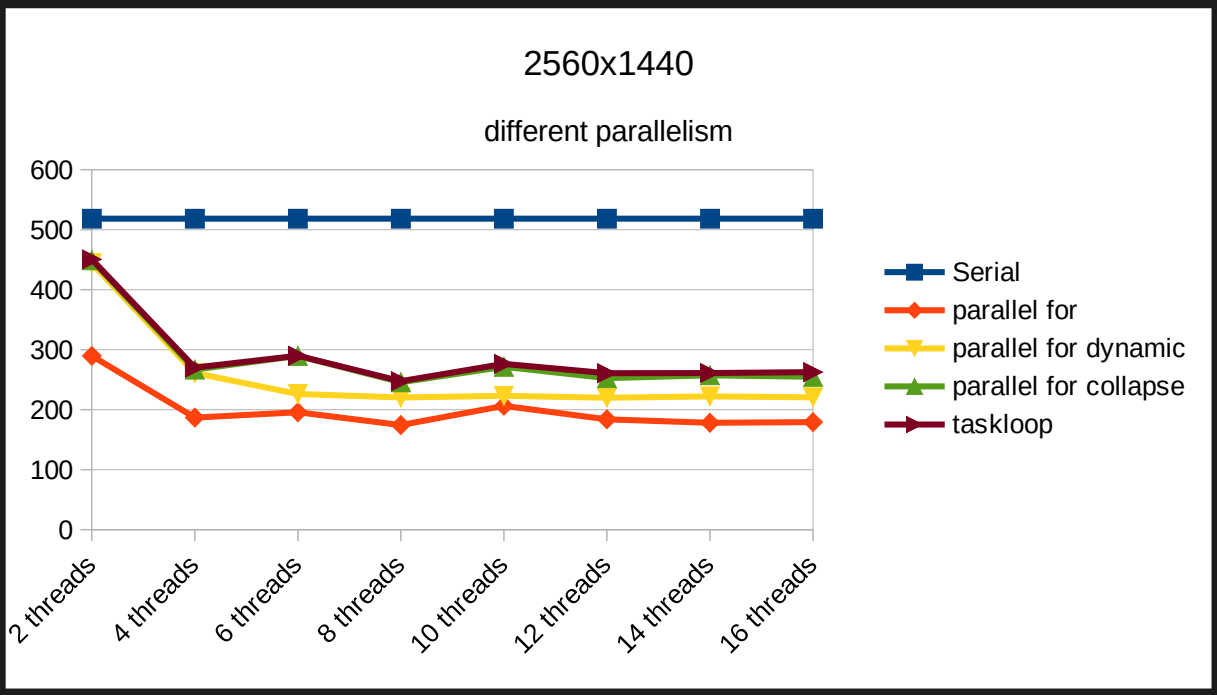
Parallelism	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
Serial	21.8	21.8	21.8	21.8	21.8	21.8	21.8	21.8
parallel for	12.3728	7.216415	8.151925	9.13053	8.288115	7.848305	8.08104	7.21786
parallel for dynamic	23.84465	13.8245	12.757	12.1454	11.987	12.1582	12.03225	12.1504
parallel for collapse	21.69815	12.9996	14.31765	12.1175	14.3357	13.092	12.4551	11.57515
taskloop	21.8384	12.1669	14.4651	12.23775	14.9045	12.843	13.11225	13.5336



Parallelism Comparison (cont.)

2560x1440

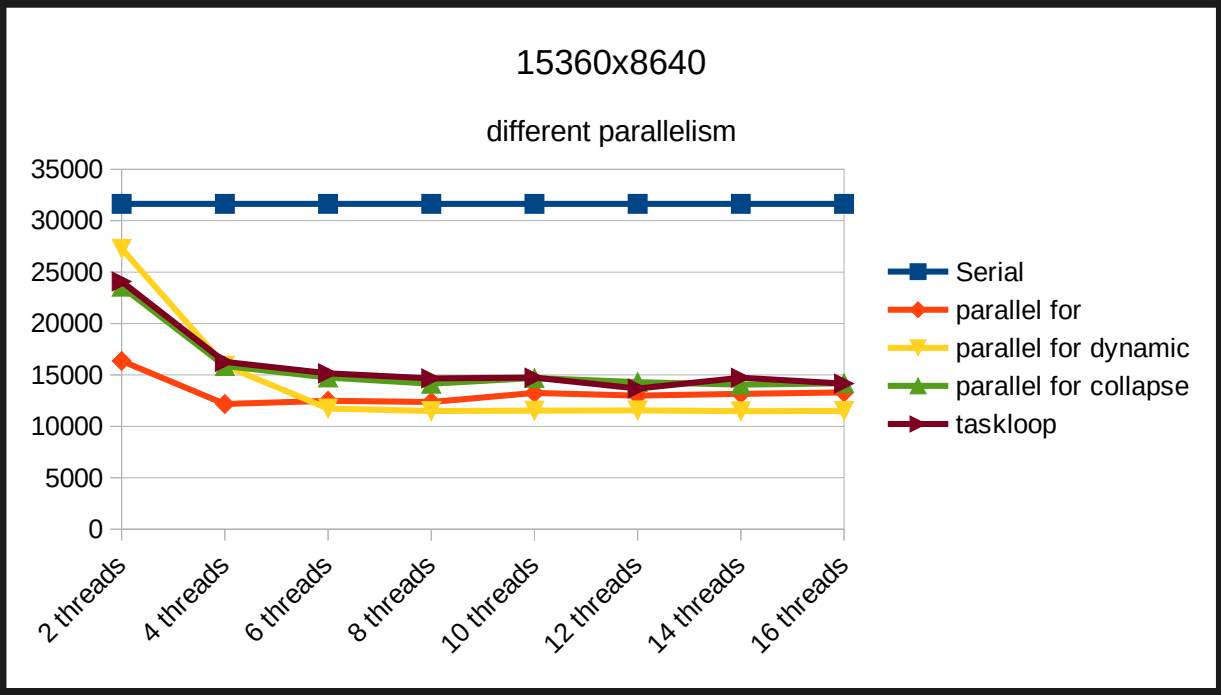
Parallelism	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
Serial	518.4	518.4	518.4	518.4	518.4	518.4	518.4	518.4
parallel for	289.793	186.9385	195.7065	174.4685	206.5495	184.0395	178.1915	179.314
parallel for dynamic	444.281	261.1385	226.215	220.394	223.089	220.011	222.176	220.7885
parallel for collapse	449.1775	266.8975	289.6915	245.679	271.481	252.682	257.413	254.7425
taskloop	450.722	270.163	290.031	247.6295	276.4785	260.821	261.1705	262.6315



Parallelism Comparison (cont.)

15360x8640

Parallelism	2 threads	4 threads	6 threads	8 threads	10 threads	12 threads	14 threads	16 threads
Serial	31632.7	31632.7	31632.7	31632.7	31632.7	31632.7	31632.7	31632.7
parallel for	16359	12171.7	12485.6	12360.65	13242.95	12989.35	13155.8	13304.95
parallel for dynamic	27274.45	15904.6	11733.6	11484.1	11528.1	11557.8	11464.6	11498.95
parallel for collapse	23587.65	15872	14742	14139.65	14691.35	14300.45	14054.95	14188.5
taskloop	24091.5	16267.65	15148.55	14657	14741.7	13676.65	14731.8	14159.55



CUDA

Hardware:

Kernel: 5.15.29 x86_64

CPU: Intel i7-4710HQ (8) @ 3.500GHz

GPU: Intel 4th Gen Core Processor

GPU: NVIDIA GeForce GTX 850M

Memory: 12 GB

Disk: SATA SSD

Compiler: gcc (GCC) 10.3.0

```
Every 2.0s: nvidia-smi

Mon Jul  4 16:34:39 2022
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 510.39.01    Driver Version: 510.39.01    CUDA Version: 11.6     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  Off  | 00000000:01:00.0 Off |                  N/A |
| N/A   52C    P0    N/A /  N/A |     33MiB /  2048MiB |     44%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
|    0   N/A  N/A      1219      G   ...-xorg-server-21.1.3/bin/X        2MiB |
|    0   N/A  N/A     83858      C   ./build/src/Main                   27MiB |
```

Problem Size:

640x360
1280x720
1920x1080
2560x1440
3840x2160
7680x4320
15360x8640

Size	Size increase	Exec time (ms)
640x480	1x	4.0832928
1280x720	2x	13.24696
1920x1080	3x	29.1912545
2560x1440	4x	54.8183428
3840x2160	6x	128.8317215
7680x4320	12x	650.753351
15360x8640	24x	NA

CUDA

Size	Size increase	Exec time (ms)
640x480	1x	21.8
1280x720	2x	80.3
1920x1080	3x	244.6
2560x1440	4x	518.4
3840x2160	6x	806.2
7680x4320	12x	5941.1
15360x8640	24x	31632.7

Serial

Results:

Using OpenMP and its directives, we achieved speedups up to 4. This speed was gained without much tinkering with the code. All we did was add some directives to the serial implementation.

On the specified hardware, 8 threads on average, seem to be the optimal number of threads. The number 8 corresponds to the number of CPU cores on the specified hardware.

`parallel for` seems to be the way to go for the simple task of iterating over all pixels of the image. For huge input sizes however, `dynamic schedule` yields better results.

Using CUDA we were able to achieved speedups up to 10. In contrast to OpenMP, we had to change the code quite a bit to be able to run it efficiently on a GPU. The small VRAM on the GPU caused some problems when trying to allocate memory for huge inputs sizes.