Parallel Kernel Image Processing Report

Michael Bartoloni
Matricola 7152862

Abstract

The purpose of this project is an exercise in parallel programming techniques using the CUDA framework. The project features the development of two kernel processing methods for a dataset of 2K resolution images, first using a sequential C++ implementation, then using an optimized parallel version using CUDA, and a comparison between sequential and parallel execution times.

1. Introduction

Kernel image processing is a fundamental technique in image processing and computer vision, used for tasks such as filtering, edge detection, and noise reduction.
A separable kernel and a nonseparable kernel were tested in a sequential and in a highly parallel environment. The purpose is to test the improvement in performance with different complexities of the algorithm. The benchmark is performed on 3 differently sized images. The parallel benchmark measures the execution of the kernels for increasing block sizes, comparing them with the sequential time.

1.1. The Images

The three images represent different resolutions to test the algorithms on. They go from low to high resolution, as the resolutions scale from 200x300 for the first image, 2K for the second image, and 8K for the last image. This aims to find possibly different behaviors of the algorithms when applied to different magnitudes of data.

1.2. Kernel Image Processing

Kernel image processing involves applying a convolution filter to an image. In our case, the convolution is performed using a range of increasingly sized kernels (3x3 to 15x15, with a stride of 2 in the kernel's width). The convolution operation involves sliding the kernel over the image and computing a weighted sum of pixel values within the kernel window. Although separable kernels can reduce the computational cost by breaking a 2D filter into two 1D filters, the nonseparable version processes a quadratic number (with respect to kernel's width) per convolution. Both versions will be evaluated, in hope to provide insight into the performance gains achievable when parallelizing computationally intensive tasks on GPUs. In [Figure 1], a visualization of the task is shown.

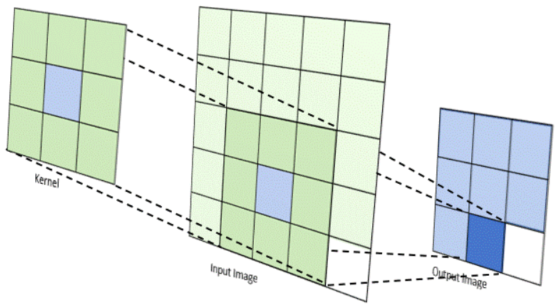

Figure 1. Convolution visualization

1.3. The Algorithm

The algorithm performs a convolution operation, a data-intensive process in which each output pixel is calculated as a weighted sum of a neighborhood of input pixels. This requires management of memory accesses, particularly because the convolution window must be applied across the entire image, including its borders.
The separable kernel applies a Gaussian blur to the image, and the nonseparable kernel applies an image detection filter (Laplacian of Gaussian). The implementations are shown in [Source code 1 and 2] and, as shown, the choice for handling the boundary conditions is to clamp the pixel indices to valid ranges, so that each out-of-range pixel index is translated to the closest valid index. Examples of how the filters work on the low resolution image are shown in [Figures 2, 3, 4].

```
void separableConvolution(const unsigned char*
    input, unsigned char* output,
                               int width, int height
    ,
                              const float* kernel,
    int kernelRadius) {
    float* temp = new float[width * height]();
    // Horizontal pass
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float sum = 0.0f;
            for (int k = -kernelRadius; k <=
kernelRadius; k++) {
                int xx = std::min(std::max(x +
k, 0), width - 1);
                sum += kernel[kernelRadius + k]
 * input[y * width + xx];
            }
            temp[y * width + x] = sum;
        }
    }
    // Vertical pass
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float sum = 0.0f;
            for (int k = -kernelRadius; k <=
kernelRadius; k++) {
                int yy = std::min(std::max(y +
k, 0), height - 1);
                sum += kernel[kernelRadius + k]
 * temp[yy * width + x];
            }
            int val = static_cast<int>(sum);
            output[y * width + x] = static_cast
<unsigned char>(std::min(std::max(val, 0),
255));
        }
    }
    delete[] temp;
}
```

Source code 1. Separable convolution

```
void nonSeparableConvolution(const unsigned
    char* input, unsigned char* output,
                              int width, int
    height,
                              const float*
    kernel, int kernelSize) {
    int kernelRadius = kernelSize / 2;
    for (int y = 0; y < height; y++) {
        for (int x = 0; x < width; x++) {
            float sum = 0.0f;
            for (int ky = -kernelRadius; ky <=
kernelRadius; ky++) {
                for (int kx = -kernelRadius; kx
 <= kernelRadius; kx++) {
                    int yy = std::min(std::max(
y + ky, 0), height - 1);
                    int xx = std::min(std::max(
x + kx, 0), width - 1);
                    sum += kernel[(ky +
kernelRadius) * kernelSize + (kx +
kernelRadius)] * input[yy * width + xx];
                }
            }
            int val = static_cast<int>(sum);
            output[y * width + x] = static_cast
<unsigned char>(std::min(std::max(val, 0),
255));
        }
    }
}
```

Source code 2. Nonseparable Convolution

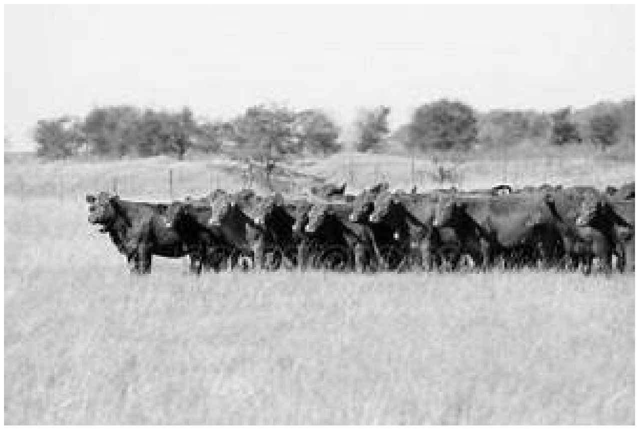
Figure 2. Starting low resolution image

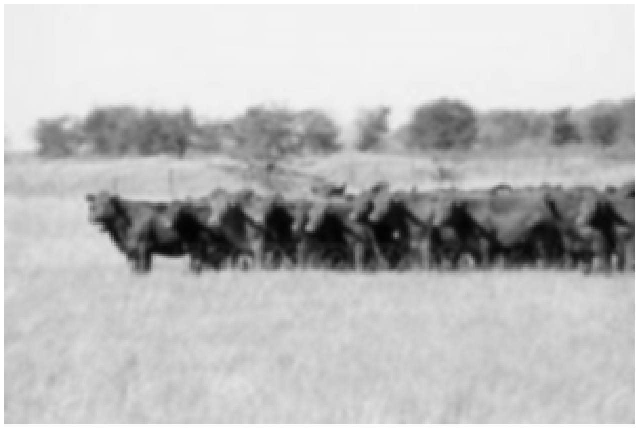
Figure 3. Gaussian blur filter

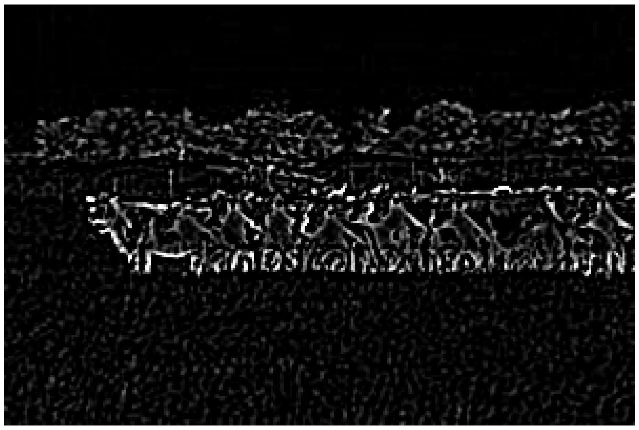
Figure 4. Edge detection filter

2. Sequential Benchmark

The sequential benchmark loads the three images in a grayscale format and applies the separable kernel and the

nonseparable kernel on the images, for each kernel size (7 in total, starting from kernel width of 3 with a step of 2).
To mitigate variability in performance measurements, each kernel is executed 100 times, and the average elapsed time is computed for a more stable benchmark.

2.1. Results

The results [Tables 1, 2, 3] show a marked difference in execution times between the separable and nonseparable kernel implementations. As expected, the more complex nonseparable kernel runs slower, since it requires more computation in comparison with the separable kernel.
As we would expect, more kernel width equals more difference in execution times, as the scaling of the number of operations is different.

Kernel size	Separable time	Nonseparable time
3	0.418 (ms)	0.822 (ms)
5	0.734 (ms)	1.824 (ms)
7	1.035 (ms)	3.323 (ms)
9	1.253 (ms)	5.436 (ms)
11	1.397 (ms)	7.696 (ms)
13	2.041 (ms)	10.539 (ms)
15	2.039 (ms)	13.636 (ms)

Table 1. Sequential execution times for Low image (displayed in ms)

Kernel size	Separable time	Nonseparable time
3	0.019 (s)	0.039 (s)
5	0.034 (s)	0.084 (s)
7	0.044 (s)	0.149 (s)
9	0.061 (s)	0.253 (s)
11	0.070 (s)	0.353 (s)
13	0.089 (s)	0.477 (s)
15	0.101 (s)	0.621 (s)

Table 2. Sequential execution times for Medium image (displayed in s)

Kernel size	Separable time	Nonseparable time
3	0.310 (s)	0.489 (s)
5	0.524 (s)	1.085 (s)
7	0.633 (s)	1.927 (s)
9	0.893 (s)	3.292 (s)
11	1.091 (s)	4.624 (s)
13	1.438 (s)	6.266 (s)
15	1.713 (s)	8.241 (s)

Table 3. Sequential execution times for High image (displayed in s)

3. Parallel Benchmark

The parallel benchmark features an optimized version of the algorithm, executed on an NVIDIA GeForce RTX 3060 (12GB). Three versions of the algorithm, using different memory exploiting, are evaluated using increasing block sizes, ranging from 32 to 1024 threads per block. We do not need to test beyond this range because the SM can manage a maximum of 1536 threads.
The first implementation of the kernels [Source code 3 and 4] uses only global memory accesses, so the next section will cover some optimizations that can be made to speed up the processing. Note that, in order to repeat the separable convolution's two-pass approach on a multithreaded algorithm, we actually need some synchronization after storing the intermediate results in the *temp* buffer, losing some performance, but this loss is necessary to ensure correctness.

```
__global__ void separableConvolutionKernelGlobal(const unsigned char* input, unsigned char* output,
    float* temp, int width, int height, float* kernel, int kernelRadius)
{
    int tx = threadIdx.x;
    int index = blockIdx.x * blockDim.x + tx;

    if (index < width * height) {
        int x = index % width;
        int y = index / width;

        float sum = 0.0f;
        for (int kx = -kernelRadius; kx <= kernelRadius; kx++) {
            int globalX = clamp_int(x + kx, 0, width - 1);
            int globalIndex = y * width + globalX;
            sum += kernel[kx + kernelRadius] * float(input[globalIndex]);
        }
        temp[index] = sum;
        __syncthreads();
        sum = 0.0f;
        for (int ky = -kernelRadius; ky <= kernelRadius; ky++) {
            int globalY = clamp_int(y + ky, 0, height - 1);
            int tempIndex = globalY * width + x;
            sum += kernel[ky + kernelRadius] * temp[tempIndex];
        }
        output[index] = static_cast<unsigned char>(clamp_int(static_cast<int>(sum), 0, 255));
    }
}
```

Source code 3. Separable convolution with only global memory usage

```
__global__ void nonSeparableConvolutionKernelGlobal(const
    unsigned char* input, unsigned char* output
    ,
    int width, int height, float* kernel, int
    kernelRadius, int kernelWidth) {
    int tx = threadIdx.x;
    int index = blockIdx.x * blockDim.x + tx;

    if (index < width * height) {
        int x = index % width;
        int y = index / width;

        float sum = 0.0f;
        for (int ky = -kernelRadius; ky <=
    kernelRadius; ky++) {
            for (int kx = -kernelRadius; kx <=
    kernelRadius; kx++) {
                int globalX = clamp_int(x + kx,
     0, width - 1);
                int globalY = clamp_int(y + ky,
     0, height - 1);
                int globalIndex = globalY *
    width + globalX;
                sum += kernel[(ky +
    kernelRadius) * kernelWidth + (kx +
    kernelRadius)] * float(input[globalIndex]);
            }
        }
        output[index] = static_cast<unsigned
    char>(clamp_int(static_cast<int>(sum), 0,
    255));
    }
}
```

Source code 4. Nonseparable convolution with only global memory usage

3.1. Optimizations

3.1.1 Constant Memory

Constant memory is very useful and fast for data that is frequently accessed but never changed during the execution of the algorithm, as it has a much faster access time compared to global memory (when not cached). Loading the kernels onto them is always a very good idea, as this is a very easy upgrade to the access patterns of our algorithm which comes at practically no cost. [Source code 5] shows how the constant kernels are declared and allocated. Since the largest kernels have a width of 15, we always declare maximum sized kernels and allocate what we need to use at runtime. [Source code 6 and 7] show how to change the global kernel to use constant memory (which is only changing the variable we access when computing the convolution).

```
__constant__ float cKernel[15];
__constant__ float cKernelNonSeparable[225];


int main{
    //...
    cudaCheckError(cudaMemcpyToSymbol(cKernel,
    separableKernels[j], kernelSizeInBytes));
    //...
    cudaCheckError(cudaMemcpyToSymbol(
    cKernelNonSeparable, nonSeparableKernels[j
    ], kernelSizeInBytes));
}
```

Source code 5. Loading kernels in the shared memory

```
__global__ void
    separableConvolutionKernelConstant(const
    unsigned char* input, unsigned char* output
    ,
    float* temp, int width, int height, int
    kernelRadius)
{
    //...

    if (index < width * height) {
        //...
        float sum = 0.0f;
        for (int kx = -kernelRadius; kx <=
    kernelRadius; kx++) {
            //...
            sum += cKernel[kx + kernelRadius] *
     float(input[globalIndex]);
        }
        temp[index] = sum;
        __syncthreads();
        sum = 0.0f;
        for (int ky = -kernelRadius; ky <=
    kernelRadius; ky++) {
            //...
            sum += cKernel[ky + kernelRadius] *
     temp[tempIndex];
        }
        //...
    }
}
```

Source code 6. Separable kernel using constant memory

```
__global__ void
    nonSeparableConvolutionKernelConstant(const
     unsigned char* input, unsigned char*
    output,
    //...

    if (index < width * height) {
        //...
        float sum = 0.0f;
        for (int ky = -kernelRadius; ky <=
    kernelRadius; ky++) {
            for (int kx = -kernelRadius; kx <=
    kernelRadius; kx++) {
                //...
                sum += cKernel[(ky +
    kernelRadius) * kernelWidth + (kx +
    kernelRadius)] * float(input[globalIndex]);
            }
        }
        //...
    }
}
```

Source code 7. Nonseparable kernel using constant memory

3.1.2 Shared memory

For both the previous implementations, the convolution operation is performed on data that resides in the global memory. This could potentially bring to a loss of performance, especially if data is swapped back and forth across global memory and caches, and shared memory (in this particular case, applying tiling) could help reducing the memory overhead and speed up the whole process. [Source code 8 and 9] show an implementation of the convolutions for both separable and nonseparable kernels. Now, both kernels require to load a chunk of the image (enlarged with a halo, its width depending on the kernel's radius) into the block's shared memory in order to use it to perform the convolutions. The design choice for the loading of the tile sees each thread loading more than one pixel on average, by linearizing thread's indexes and distributing the work uniformly. This adds a layer of synchronization to both types of kernels, necessary to wait for all the threads to load the pixels of the tile before calculating the convolution.
The rest of the computation is basically the same for the nonseparable kernel, the only change is the use of shared memory in combination of constant memory when calculating the convolution. For the separable kernel, however, there's still a problem to take care of. The *temp* buffer, used to store the intermediate results from the horizontal pass, is still in global memory and the vertical pass uses that same buffer, so we have to optimize the memory usage to use as much shared memory as possible, but there are a couple of considerations:

- In global and constant memory settings, the buffer is accessible by all threads independently from their block assignment, which is a crucial condition for cooperation among threads. Shared memory, unfortunately, does not have such properties, and is accessible only from threads within the same block.
- To calculate the vertical pass, we need not only the tile's pixel but also the vertical halo's pixels, so as before we have to load more pixels than the actual number of threads in the block, but the enlargement factor is a little less, since can avoid to load the horizontal halo.

The solution is similar to loading a tile (the logic is basically the same), but instead of loading a pixel, we perform the horizontal pass using the previously loaded shared tile and store the intermediate results. Since it's possible to declare only one shared variable for kernel and the two tiles have different formats, the launch parameter for the shared memory size is the sum of the two tiles, plus eventual padding after the *unsigned char* image tile, permitting the *float* intermediate results tile to be reinterpret casted from the shared buffer. The synchronization after the horizontal pass remains, as it's also needed for correct initialization of the shared *temp* buffer.
With these approaches, we expect an improvement in performance, given the faster memory type, but synchronization and the added instructions per thread could damage the theoretical speedup.

```
__global__ void separableConvolutionKernelConstAndShared(const unsigned char* input, unsigned char* output,
    int width, int height, int kernelRadius, int blockWidth, int blockHeight, int sharedTileWidth, int sharedTileHeight, int enlargementFromTile, int enlargementFromTemp)
{
    int tx = threadIdx.x;

    int tileX = blockIdx.x % cNTilesX;
    int tileY = blockIdx.x / cNTilesX;
    int tileOriginX = tileX * blockWidth;
    int tileOriginY = tileY * blockHeight;

    int totalSharedPixelsTile = sharedTileWidth * sharedTileHeight;

    extern __shared__ unsigned char sharedBuffer[];
    float* tempBuffer = reinterpret_cast<float*>(sharedBuffer + totalSharedPixelsTile * sizeof(unsigned char));

    for (int i = 0; i < enlargementFromTile; i++) {
        int linearIndex = tx + i * blockDim.x;
        if (linearIndex < totalSharedPixelsTile) {
            int x = linearIndex % sharedTileWidth;
            int y = linearIndex / sharedTileWidth;
            int globalX = tileOriginX - kernelRadius + x;
            int globalY = tileOriginY - kernelRadius + y;
            int globalIndex = globalY * width + globalX;
            if (globalX >= 0 && globalX < width && globalY >= 0 && globalY < height) {
                sharedBuffer[linearIndex] = input[globalIndex];
            } else {
                int clampedGlobalX = clamp_int(globalX, 0, width - 1);
                int clampedGlobalY = clamp_int(globalY, 0, height - 1);
                int clampedGlobalIndex = clampedGlobalY * width + clampedGlobalX;
                sharedBuffer[linearIndex] = input[clampedGlobalIndex];
            }
        }
    }
    __syncthreads();
    for (int i = 0; i < enlargementFromTemp; i
++) {
        int linearIndex = tx + i * blockDim.x;
        if (linearIndex < sharedTileHeight *
blockWidth) {
            int x = linearIndex % blockWidth;
            int y = linearIndex / blockWidth;
            tempBuffer[linearIndex] = 0.0f;
            int sharedTileIndex = y *
sharedTileWidth + x + kernelRadius;
            for (int kx = -kernelRadius; kx <=
kernelRadius; kx++) {
                if (sharedTileIndex >= 0 &&
sharedTileIndex < totalSharedPixelsTile) {
                    tempBuffer[linearIndex] +=
cKernel[kx + kernelRadius] * float(
sharedBuffer[sharedTileIndex + kx]);
                }
            }
        }
    }
    __syncthreads();
    int localX = tx % blockWidth;
    int localY = tx / blockWidth;
    int globalX = tileOriginX + localX;
    int globalY = tileOriginY + localY;

    if (globalX < width && globalY < height) {
        float sum = 0.0f;
        for (int ky = -kernelRadius; ky <=
kernelRadius; ky++) {
            int tempIndex = (localY +
kernelRadius + ky) * blockWidth + localX;
            sum += cKernel[ky + kernelRadius] *
 tempBuffer[tempIndex];
        }
        output[globalY * width + globalX] =
static_cast<unsigned char>(clamp_int(
static_cast<int>(sum), 0, 255));
    }
}
```
Source code 8. Separable kernel using constant and shared memory

```
__global__ void
    nonSeparableConvolutionKernelConstAndShared
    (const unsigned char* input, unsigned char*
     output,
    int width, int height, int kernelRadius,
    int blockWidth, int blockHeight, int
    sharedTileWidth, int sharedTileHeight, int
    enlargement) {

    int tx = threadIdx.x;

    int tileX = blockIdx.x % cNTilesX;
    int tileY = blockIdx.x / cNTilesX;
    int tileOriginX = tileX * blockWidth;
    int tileOriginY = tileY * blockHeight;

    int totalSharedPixels = sharedTileWidth *
    sharedTileHeight;

    //Same tiling as the separable version...

    int localX = tx % blockWidth;
    int localY = tx / blockWidth;
    int globalX = tileOriginX + localX;
    int globalY = tileOriginY + localY;

    if (globalX < width && globalY < height) {
        float sum = 0.0f;
        for (int ky = -kernelRadius; ky <=
    kernelRadius; ky++) {
            for (int kx = -kernelRadius; kx <=
    kernelRadius; kx++) {
                int sharedTileIndex = (localY +
     kernelRadius + ky) * sharedTileWidth +
    localX + kernelRadius + kx;
                sum += cKernelNonSeparable[(ky
    + kernelRadius) * (2 * kernelRadius + 1) +
    kx + kernelRadius] *
                       float(tileBuffer[
    sharedTileIndex]);
            }
        }
        output[globalY * width + globalX] =
    static_cast<unsigned char>(clamp_int(
    static_cast<int>(sum), 0, 255));
    }
}
```
Source code 9. Nonseparable kernel using constant and shared memory

3.2. Results

Given the number of variables in this test, an attempt to reduce the dimensionality of the results, in order to have comprehensible plots and tables (a reasonable number of them), was to identify the best block sizes among those that fully utilize the block structure of the GPU (i.e. not sizes 32 and 1024) for each combination of memory type and block size, select a single block size per image resolution, and then reason on those results only. A block size was chosen as best for an image based on how many times it was the fastest layout for the kernel, compared to other block sizes. [Tables 7, 8, 9] show the maximum time difference for each memory type from the most performing block size and the other reasonable block sizes. In [Figures 5, 6, 7], execution times are plotted for each *(memory type, kernel type)* pair. For the nonseparable kernels, the results behave as expected, showing gradual improvement as the optimizations become more aggressive. For the separable kernels, things get strange when looking at shared memory performance, as it's by far the worst compared to separable kernel alternatives.

4. Conclusions

The parallel benchmark results clearly demonstrate that the use of GPUs for kernel image processing always leads to substantial performance improvements over a sequential approach. The highly parallel nature of the GPU enables a dramatic reduction in execution times.
The kernels incur different computational overheads, due to their different complexity in processing the convolution:

Table 4. Times (ms) and speedups for low resolution image (block=128)

Mode	global				constant				shared			
KernelType	separable		nonseparable		separable		nonseparable		separable		nonseparable	
Metric	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup
KernelSize												
3	0.00569	73.4	0.00595	138	0.00556	75.2	0.00581	141	0.00741	56.4	0.00674	122
5	0.00616	119	0.00943	194	0.00604	122	0.00895	204	0.00827	88.7	0.00865	211
7	0.00711	146	0.0128	260	0.00698	148	0.012	278	0.00936	111	0.0108	307
9	0.0073	172	0.0188	289	0.00717	175	0.0178	306	0.0102	123	0.0135	403
11	0.0082	170	0.0241	319	0.00805	174	0.0228	337	0.0119	117	0.0171	450
13	0.0084	243	0.033	319	0.0083	246	0.0312	338	0.0133	153	0.0211	500
15	0.00935	218	0.0406	336	0.0092	222	0.0379	360	0.0147	139	0.0259	526

Table 5. Times (ms) and speedups for medium resolution image (block=256)

Mode	global				constant				shared			
KernelType	separable		nonseparable		separable		nonseparable		separable		nonseparable	
Metric	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup
KernelSize												
3	0.138	138	0.107	367	0.136	139	0.103	380	0.164	116	0.143	273
5	0.176	194	0.257	326	0.175	196	0.234	358	0.191	179	0.215	390
7	0.179	246	0.39	383	0.175	252	0.362	412	0.218	202	0.294	508
9	0.226	271	0.653	388	0.228	270	0.602	420	0.251	244	0.402	630
11	0.251	278	0.878	402	0.256	272	0.806	438	0.281	248	0.538	656
13	0.278	319	1.25	380	0.276	321	1.16	410	0.306	290	0.694	687
15	0.307	330	1.6	388	0.302	335	1.45	429	0.347	292	0.894	695

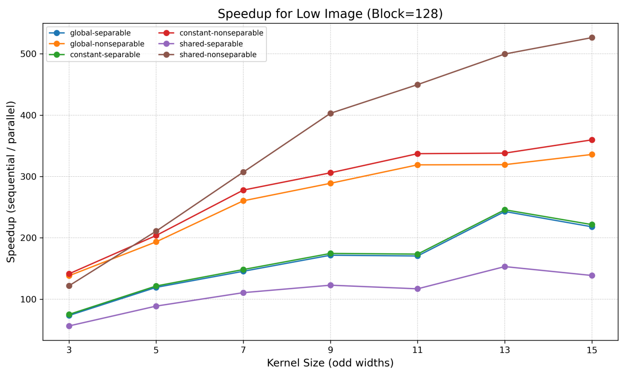

Figure 5. Speedups for the low resolution image

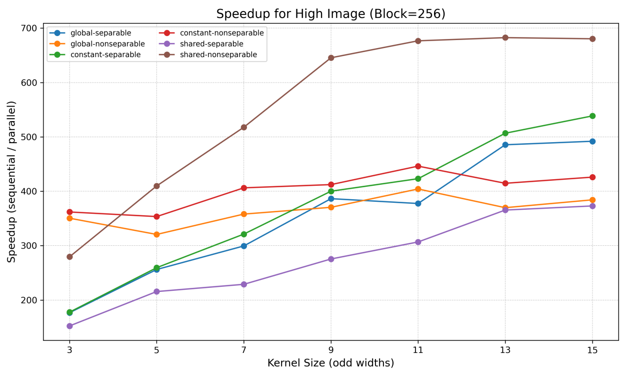

Figure 7. Speedups for the high resolution image

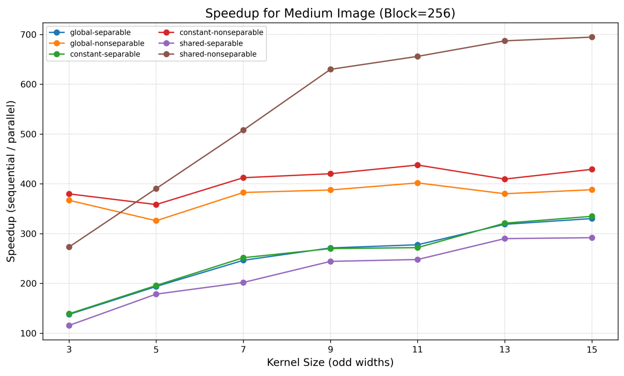

Figure 6. Speedups for the medium resolution image

even if a thread elaborates only a pixel, the number of operations is linear for the separable kernel and quadratic for the nonseparable kernel (with respect to the kernel's width). Despite these differences, both implementations show massive overall improvements when executed in parallel, and their differences are mitigated.

The nonseparable kernel behaves as expected: each memory optimization provides some upgrade to performance, and the shared memory version performs best (for a kernel size that is not very small). However, when it comes to the separable kernels, shared memory performs way worse than even global memory. Global and constant memory behave almost identical on low and medium resolutions. The shared memory's results may be caused by the added synchronization and operations per thread, which are in part not present and may be negligible for a quadratic cost operation like a nonseparable convolution, but have an impact on performance when there are less operations, and the results confirm it. For constant and global memory's similarity, probably with a low and medium resolution image, the memory traffic is not so high and the kernels for each block work stay most of the time in the cache, even when using only global memory accesses, leading to similar results. However, when the image gets bigger, cache misses

Table 6. Times (ms) and speedups for high resolution image (block=256)

Mode	global				constant				shared			
KernelType	separable		nonseparable		separable		nonseparable		separable		nonseparable	
Metric	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup	time (ms)	speedup
KernelSize												
3	1.75	177	1.4	350	1.74	178	1.35	362	2.03	152	1.75	279
5	2.05	256	3.39	321	2.02	259	3.07	353	2.43	216	2.65	410
7	2.11	299	5.38	358	1.97	321	4.74	406	2.76	229	3.72	518
9	2.31	386	8.89	370	2.23	400	7.98	412	3.24	275	5.1	645
11	2.89	378	11.4	404	2.58	423	10.4	446	3.56	307	6.83	677
13	2.96	486	16.9	370	2.84	507	15.1	415	3.93	365	9.18	683
15	3.48	492	21.4	384	3.18	539	19.3	426	4.59	373	12.1	680

Table 7. Max time difference between the most performing block size and the other sizes (ms) per memory type for the low resolution image

Mode	global	constant	shared
KernelSize			
3	0.001	0.001	0.001
5	0.000	0.000	0.001
7	0.001	0.000	0.001
9	0.001	0.001	0.001
11	0.001	0.000	0.003
13	0.001	0.001	0.003
15	0.002	0.001	0.004

Table 8. Max time difference between the most performing block size and the other sizes (ms) per memory type for the medium resolution image

Mode	global	constant	shared
KernelSize			
3	0.048	0.048	0.047
5	0.062	0.063	0.071
7	0.064	0.067	0.082
9	0.044	0.043	0.074
11	0.065	0.057	0.170
13	0.052	0.053	0.199
15	0.037	0.039	0.227

Table 9. Max time difference between the most performing block size and the other sizes (ms) per memory type for the high resolution image

Mode	global	constant	shared
KernelSize			
3	0.680	0.654	0.696
5	0.257	0.180	0.938
7	0.395	0.402	1.109
9	0.208	0.185	0.966
11	0.098	0.472	2.894
13	0.286	0.280	2.606
15	0.120	0.228	2.887

happen more frequently, and constant memory, which is always accessible with cache speed, has the edge over a full global memory approach.