ECE415 - High Performance Computing

– Homework 3 –

Introduction to CUDA: Accelerating Convolutions

submitted by

Aggeliki-Ostralis Vliora, Spyridon Liaskonis

January 19, 2025

Aggeliki-Ostralis Vliora, Spyridon Liaskonis
avliora@uth.gr, sliaskonis@uth.gr
Student ID: 03140,03381

Contents

1 Introduction **1**

2 Proccess **1**

2.1 DeviceQuery Output 1

2.2 Nvcc Compiler Parameters 1

2.3 CUDA implementation using one 2D Thread Block 2

2.3.1 Evaluation 2

2.4 CUDA implementation using a 2D grid of 2D Thread Blocks 4

2.4.1 Evaluation 5

2.5 Evaluation using doubles instead of floats 8

2.6 Evaluation Questions 11

2.6.1 Frequency of Input Photo and Filter's Elements Access 11

2.6.2 Ratio of Memory Accesses to Floating Point Operations . . . 13

2.7 Solving Divergence Problem 13

2.7.1 Implementation 13

2.7.2 Evaluation 15

A Device Query Outputs **17**

A.0.1 Mars 17

A.0.2 Venus 22

1 Introduction

In this project, we implement a two-dimensional convolution filter by decomposing it into two one-dimensional operations: a row-wise and a column-wise convolution. These operations are applied separately to a random array or image. More specifically, we first apply the row convolution on an initial input photo and store the result in a buffer. Then using the buffer as an input photo we apply the column convolution which gives us the final result (same as applying the 2D filter).

We implement the row and column convolution algorithms as different functions/kernels and execute them on a CPU and GPU. Furthermore, we experiment with different grid/block geometries for the GPU execution, we analyze the performance of each execution type by comparing them, we compare and experiment with the accuracy of each type of execution, and finally we work on an optimization for both CPU and GPU codes.

2 Proccess

2.1 DeviceQuery Output

In the first step we use the deviceQuery Utility in order to get useful and detailed information about the GPU(s) of the systems that we used. For this assignment we used two different systems, referred to as Mars and Venus, used for development and evaluation respectively. The detailed deviceQuery output for both systems is provided in Appendix A.

2.2 Nvcc Compiler Parameters

Using `nvcc --help`, we observe the available parameters that nvcc provides. In order to compile the code, we used the following parameters:

```
nvcc -O4 -o Conv2d Conv2d.cu
```

Here, we use `-O4` for maximum optimization level on the host code, and also we avoided using any other flags such as `-G` since when used, they disable all optimizations applied by nvcc, resulting in lower performance (-G, and -g was avoided only in the compiled code that we performed performance analysis on).

2.3 CUDA implementation using one 2D Thread Block

For the initial CUDA implementation of separable convolutions, we use a single 2D thread block. By analyzing the sequential CPU code for performing row and column convolutions, we observe that the computation of each element in the final matrix can be done independently of the other elements.

In the CUDA implementation, each thread is assigned the task of computing the convolution for one specific element of the output matrix. Since we are using only a single block in this implementation, we set the block dimensions to match the size of the image (imageW and imageH). This ensures that the total number of threads equals the total number of elements in the output image. Listings 1 and 2 show the CUDA implementation for the row and column convolutions respectively.

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src, float *d_Filter,
                        int imageW, int imageH, int filterR) {
    int tx=threadIdx.x;
    int ty=threadIdx.y;
    float sum=0;

    if (tx < imageW && ty < imageH) {
      for (int k = -filterR; k <= filterR; k++) {
          int d = tx + k;
          if (d >= 0 && d < imageW) {
            sum += d_Src[ty * imageW + d] * d_Filter[filterR - k];
          }
      }
      d_Dst[ty * imageW + tx] = sum;
    }
  }
}
```

Listing 1: CUDA Implementation for Row Convolution

2.3.1 Evaluation

The maximum image size supported by the current implementation is 32x32. Any size bigger than that will lead to runtime errors. This happens, due to the geometry that's used, which is a single 2D block of threads. Each block has a total of 1024 threads. In our implementation, each output element of the final photo is computed by a single thread independently. Thus, the maximum number of elements that can be computed at this point is 1024, which means that the maximum image supported is 32x32.

```
__global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, float *d_Filter,
                                     int imageW, int imageH, int filterR) {
    int tx=threadIdx.x;
    int ty=threadIdx.y;
    float sum=0;

    if (tx < imageW && ty < imageH) {
      for (int k = -filterR; k <= filterR; k++) {
          int d = ty + k;
          if (d >= 0 && d < imageW) {
            sum += d_Src[d * imageW + tx] * d_Filter[filterR - k];
          }
      }
      d_Dst[ty * imageW + tx] = sum;
  }
}
```

Listing 2: CUDA Implementation for Column Convolution

In the next part of the evaluation, we test the accuracy of the results by comparing them with the results from the CPU. Our goal here is to find the maximum number of decimal point precision for different number of filter radius sizes. For each filter radius size, we compare the results with the results of the CPU and we determine the maximum decimal points supported that dont lead to precision errors. All the tests in this part were done for the maximum image size supported, 32x32.

Radius	Max Error	Number of Digits
1	0.007812	.2
2	0.015625	.1
3	0.0625	.1
4	0.1875	.0
5	0.25	.0
6	0.5	.0
7	0.375	.0
8	0.75	.0
9	1.0	2.
10	1.0	2.
11	1.5	2.
12	1.0	2.
13	1.5	2.
14	1.0	2.
15	1.0	2.

2.4 CUDA implementation using a 2D grid of 2D Thread Blocks

In this section, we try to solve the problem that we introduced with the limited geometry used in section 2.3. To solve this problem and thus be able to support larger image sizes, we change the geometry of the GPU Grid. Here, we use a 2D grid of 2D thread blocks. By changing the geometry of the grid, we are able to use more blocks and thus more threads. The idea of the second implementation is the same as the first one, each output element is calculated by one thread. With this idea in mind, we now separate the input image into multiple tiles of size `BLOCK_SIZExBLOCK_SIZE`, where block size is the maximum size of a block, which in our case is 32. Each block from the grid is now responsible for calculating the output elements of the sub-photo. With this new geometry, we only need to change the way of thread indexing from our previous CUDA implementation. The new code for both row and column convolutions is provided in Listings 3, 4 respectively.

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src, float *d_Filter,
                        int imageW, int imageH, int filterR) {
  int tx = (blockIdx.x * blockDim.x) + threadIdx.x;
  int ty = (blockIdx.y * blockDim.y) + threadIdx.y;
  float sum=0;

  if (tx < imageW && ty < imageH) {
    for (int k = -filterR; k <= filterR; k++) {
        int d = tx + k;
        if (d >= 0 && d < imageW) {
          sum += d_Src[ty * imageW + d] * d_Filter[filterR - k];
        }
    }
    d_Dst[ty * imageW + tx] = sum;
  }
}
```

Listing 3: CUDA Implementation for Row Convolution

```
__global__ void convolutionColumnGPU(float *h_Dst, float *h_Src, float *h_Filter,
                                     int imageW, int imageH, int filterR) {
    int tx = (blockIdx.x * blockDim.x) + threadIdx.x;
    int ty = (blockIdx.y * blockDim.y) + threadIdx.y;
    float sum=0;

    if (tx < imageW && ty < imageH) {
      for (int k = -filterR; k <= filterR; k++) {
          int d = ty + k;
          if (d >= 0 && d < imageW) {
            sum += h_Src[d * imageW + tx] * h_Filter[filterR - k];
          }
      }
      h_Dst[ty * imageW + tx] = sum;
  }
}
```

Listing 4: CUDA Implementation for Column Convolution

2.4.1 Evaluation

With this implementation the input image size is no longer limited by the size of the thread block. After experimenting with different input image sizes, we came to the conclusion that the only limitation on the input image size is now the global memory of our GPU. In our case, as seen in Appendix A, for the evaluation we used a Tesla K80 with 12GB (11441 MBytes) of global memory which for our algorithm is able to support image sizes of up to 16384x16384. By taking into consideration that we store 3 matrices of size `4*imageW*imageH bytes` (input matrix, buffer matrix and output matrix) into the global memory, we can easily observe how any image with higher resolution of 16384x16384 cannot fit into the global memory. For example for the next image size, 32768x32768, we get need:

$$32768 \cdot 32768 \cdot 4 = 4.294.967.296 \text{ bytes}$$

just for one matrix. For all three matrices, we need approximately 4.2*3=12.6 GB of global memory, which exceeds the global memory of our GPU.

After finding the maximum image size supported by algorithm (for our system), we investigated the relationship between the filter size and the maximum target accuracy (measured as the number of decimal digits) required for successful comparisons. We are using an image of size 1024x1024 and filters with radius equal to powers of

2. In the table below, we show the maximum error that we got for different radius sizes when comparing the results with the ones from the CPU.

Radius	Max Error	Number of Digits
1	0.007812	.2
2	0.046875	.1
4	0.25	.0
8	1.0	2.
16	3.0	2.
32	10.0	3.
64	40	3.
128	160	4.
256	640	5.

Figure 1 shows the maximum number of decimal digits that dont lead to accuracy errors as a function of the filter radius.

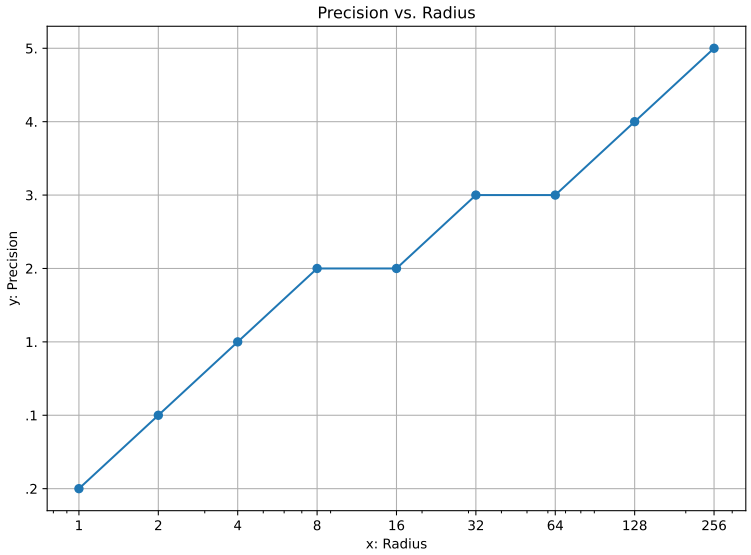

Figure 1: Maximum Decimal Point Precision for different Filter radius sizes

From Figure 1, we observe that maximum decimal point precision increases significantly as we increase the filter radius. We can see that for filters of size 4 and up to 256, we get errors beyond the decimal digits. This errors occur due to some optimizations applied by the GPU that boost performance, such as the Fused Multiply-Add (FMA). In this optimization, the GPU fuses an addition and a multiplication in one instruction and applies rounding to the result after the fusion. The CPU on

the other hand doesn't apply this optimization by default, and thus, a rounding is applied for both the addition and the multiplication separately. This, creates a rounding error between the CPU and GPU computations which increases as the number of calculations increase.

Next, we also measure the execution time of both the CPU as well as the GPU Kernel code. For the GPU, the execution times include both the execution times of the kernels as well as the times taken for transferring data from the host to device and back. In figure 2 we present the execution times from CPU and GPU code for an input image size of 16384x16384 and various filter sizes. For the execution times, we executed both codes for a certain number of times and then used the mean as the final execution time for the plots. We also include the standard deviation in Table 1. However, due to its low values, it cannot be effectively plotted alongside the execution times.

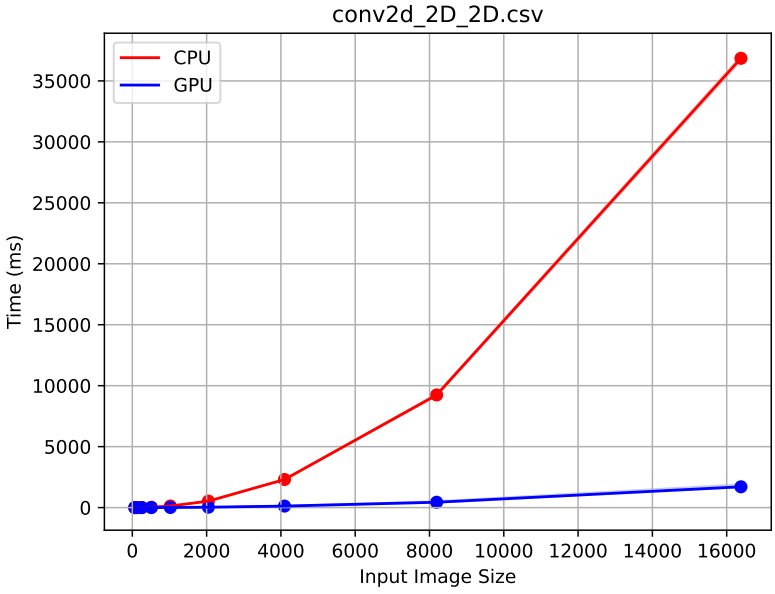

Figure 2: Cap

From Figure 2 we observe that as the size of the image increases, the computational workload also grows proportionally due to the larger number of pixels that need to be processed. As a result, the time gap between the CPU and GPU performance widens. This is because the CPU's processing time grows significantly due to its limited parallelism, while the GPU can scale its workload more effectively, maintaining relatively lower processing times.

Table 1: Standard Deviations of Execution Times for Different File Sizes

File Size	CPU Std Dev	GPU Std Dev
64	0.0269	0.0067
128	0.0391	0.0047
256	0.0620	0.0208
512	0.0701	0.0331
1024	0.1852	0.0249
2048	0.5547	0.0631
4096	9.9853	2.7877
8192	88.0053	34.3042
16384	204.1480	39.4847

2.5 Evaluation using doubles instead of floats

In this section, we repeat the previous experiment and recreate the diagrams, but instead of using floats, we use doubles. The purpose of evaluating the implementation with doubles instead of floats is to analyze the impact of numerical precision on the accuracy of the convolution operation as well as on the final total performance. Floats (single-precision) provide less precision and are more prone to rounding errors, especially in computations involving large datasets or high filter radius, as in our case. By switching to doubles (double-precision), we minimize numerical inaccuracies and observe that the increased precision positively affects the maximum error, improving accuracy by up to 7 decimal digits, as demonstrated in Figure 3. Also, Figure 4, shows the difference in decimal/integer point precision between this and the previous implementation. As expected, we get double the accuracy when we use doubles due to the double precision. The maximum error between CPU and GPU comparison for different filter radius sizes is also shown in the table below.

Radius	Max Error	Number of Digits
1	0.000000000014552	.10
2	0.000000000087311	.10
4	0.000000000582077	.9
8	0.000000001862645	.8
16	0.000000005587935	.8
32	0.000000022351742	.7
64	0.000000074505806	.7
128	0.000000298023224	.6
256	0.000001192092896	.5

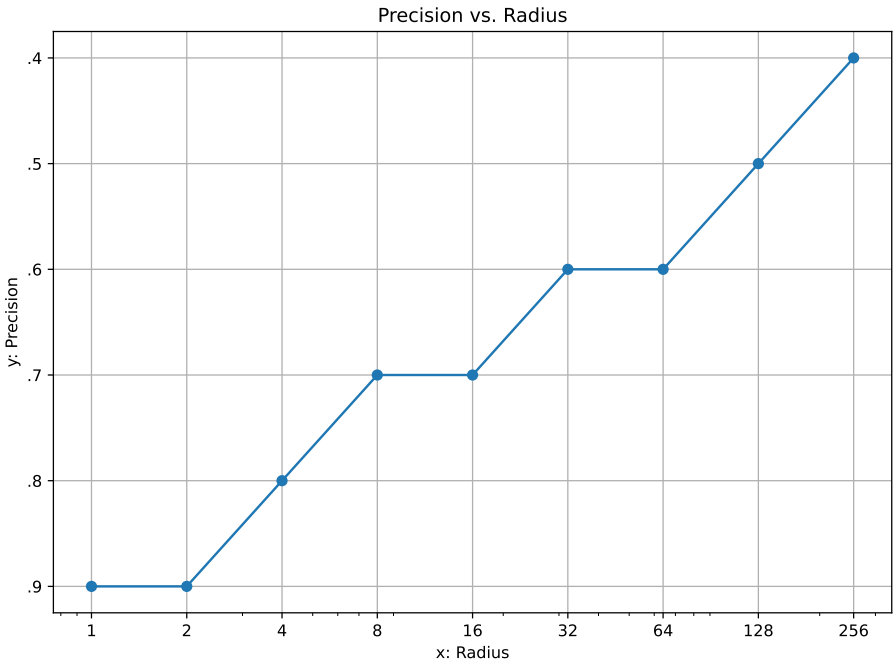

Figure 3: Maximum Decimal Point Precision for different Filter radius sizes

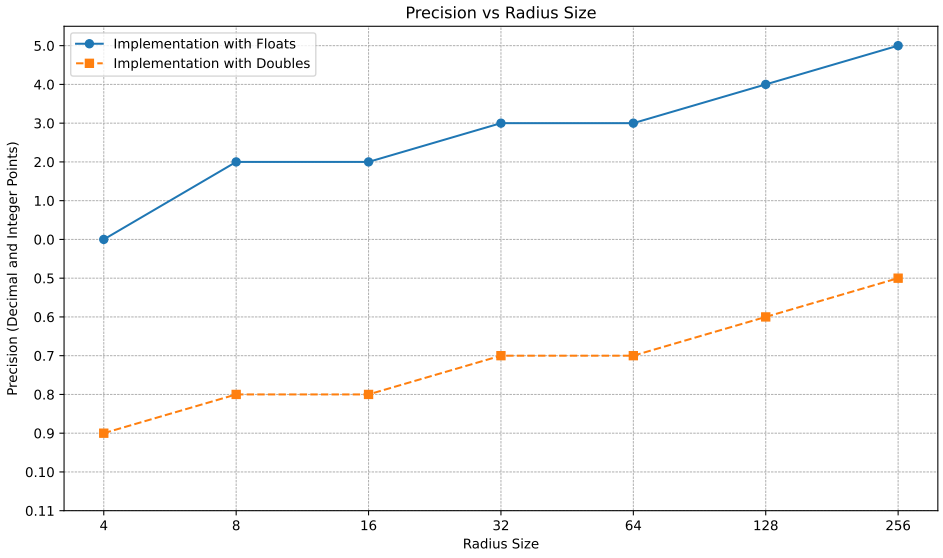

Figure 4: Precision comparison between Float And Double Implementations

When running the code on both CPU and GPU, we observed that the execution times were consistently slightly slower when using doubles compared to floats. This is because doubles require more memory and computational resources to process, resulting in higher overhead compared to floats. The difference becomes more

noticeable as the image size increases. However, the accuracy gain of double precision arithmetic, makes up for the slightly higher execution times.

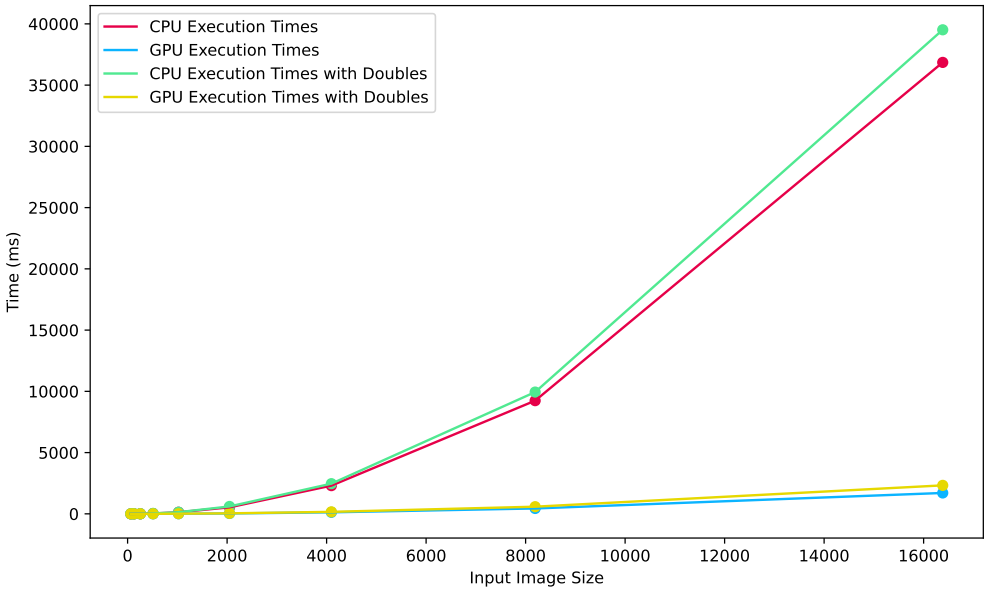

Figure 5: Execution Time Comparison between Float and Double Implementations

Table 2 shows the standard deviation from the new experiments.

Table 2: Standard Deviations for CPU and GPU Execution Times

Input Size	**CPU Std Dev**	**GPU Std Dev**
64	0.01671	0.01407
128	0.04334	0.00999
256	0.10364	0.01327
512	0.06493	0.03048
1024	0.21822	0.05488
2048	2.38910	0.09344
4096	12.19555	0.59971
8192	138.70049	30.14604
16384	354.52901	21.74190

2.6 Evaluation Questions

2.6.1 Frequency of Input Photo and Filter's Elements Access

In this section we calculate the frequency of each element from both the filter as well as the input photo.

First, we calculate the frequency of Fitler's element access. To do that, we analyze how each element from the filter is accessed in one iteration of a row convolution. Figure 6 shows the sliding of the filter through the input photo for one iteration. From the Figure, we observe that the center element of the Filter is accessed one time for each element of the input photo (thus image_width times) . The rest of the filter elements frequencies depend on their distance from the center element. Elements with distance 1 from the center will be accessed one less time than the center element, elements with distance 2 will be accessed two less times and so on. Thus, for an input photo of size imageH x imageW, the frequency of a Filter's element access as for the row convolutions is:

$$f_{row,filter}(i) = \text{imageH} \cdot (\text{imageW} - |\text{filterR - i}|), \tag{1}$$

where i, is the position of the element inside the filter, and filterR the filter radius.

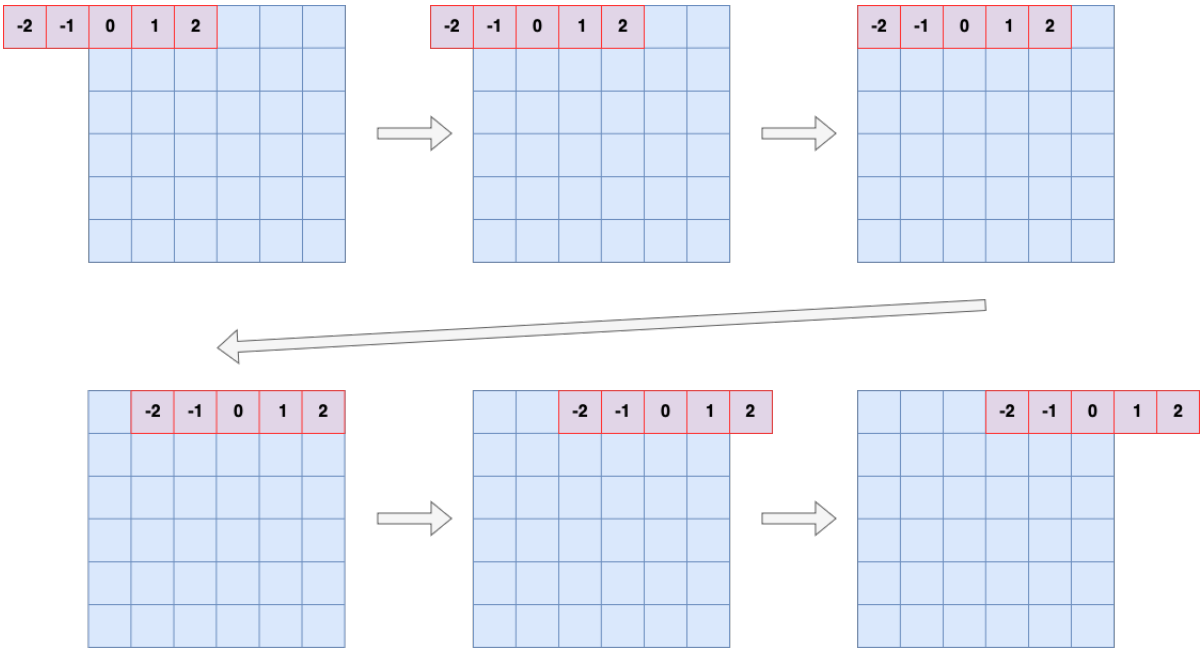

Figure 6: Row Convolution Iteration

The same formula applies to the column convolutions as well. Thus, the total frequency of a filter's element access will be:

$$f_{filter}(i) = \text{imageH} \cdot (\text{imageW} - |\text{filterR - i}|) + \text{imageW} \cdot (\text{imageH} - |\text{filterR - i}|) \quad (2)$$

In order to calculate the frequency of the input photo elements, we categorize the elements into two types:

1. Elements within the central region: These are the elements located within the boundaries of the photo, at least a distance of filterR away from the edges.
2. Boundary elements: Elements located near the edges of the photo, within a distance of filterR from the boundaries.

Elements from the first category are accessed more times than the ones from the second one, as we can observe from Figure 6. More specifically, all elements that are inside the region [filterR, imageW-filterR-1] are accessed throughout the whole row convolution for a total number of times equal to the filter length or 2*FilterR+1. As of the rest of the elements (second category), their total accesses depend on their distance from the boundaries. For example, elements that have a distance of +FilterR from the edges, are accessed one less time than the central elements, elements that have a distance of +FilterR-1 are accessed two less times and so on. The same applies for elements that are in distance of imageW-FilterR, located in the other edge of the photo. Thus, we can summarize the total number of elements access for both categories in the following equation:

$$f_{row,input}(i) = \begin{cases} 2 \cdot \text{FilterR} + 1 - (\text{FilterR - i}) & \text{, if } 0 \leq i < \text{FilterR} \\ 2 \cdot \text{FilterR} + 1 & \text{, if FilterR} \leq x \leq \text{imageW-FilterR} \\ 2 \cdot \text{FilterR} + 1 - (i - (\text{imageW - FilterR} - 1) & \text{, if i} > \text{imageW - FilterR} - 1 \end{cases}$$

The same equation applies to the column accesses as well. Thus for the total number of accesses for each input element we sum both equations and get final frequency equation:

$$f_{input}(i,j) = \begin{cases} 2 \cdot \text{FilterR} + 2 + i + j & \text{, if } 0 \leq i < \text{FilterR} \\ 4 \cdot \text{FilterR} + 2 & \text{, if FilterR} \leq x \leq \text{imageW-FilterR} - 1 \\ 2 \cdot \text{FilterR} + 2 - 2 \cdot \text{imageW} - i - j & \text{, if i} > \text{imageW - FilterR} - 1 \end{cases}$$

2.6.2 Ratio of Memory Accesses to Floating Point Operations

Each kernel (row and column convolutions) performs two floating point operations as seen in the codes in Listings 3, 4. One multiplication between an element from the input array and an element from the filter and one addition between the result of the multiplication and the previous sum:

```
sum += input[...]  * filter[...]
```

As of the memory accesses, we perform two reads from the global memory. One when reading the input element and one when reading the filter element.

Thus the ratio of memory accesses to floating point operations is $\frac{2}{2}$.

2.7 Solving Divergence Problem

2.7.1 Implementation

In our previous implementations we observe that the problem of divergence arises. Divergence refers to the situation where threads within a WARP take different execution paths. This results in wasted execution cycles and reduced performance because the threads that do not take the branch are essentially idle while the other path is executed.

In our case that happens because of boundary checks. If a thread in the image is processing a border pixel, it needs to check if the neighboring pixels are within the valid image region. This creates divergence because the threads on the border will take different execution paths compared to threads in the middle of the image, which don't need boundary checks.

So in order to eliminate that problem we get rid off this boundary check by using padding in our arrays, a technique where extra pixels (set to zero) are added around the border of the image to ensure that all threads follow the same execution path.

In our case we add padding of size filterR around the whole input image, making the new size of the image:

```
padded_image_size = (imageW+2FilterR) X (imageH+2FilterR)
```

Doing so, we can eliminate the conditional statement that causes the divergence problem. Each convolution (row and column) will begin again with the center of the filter in a border element. The convolution in these positions will execute for all elements of the filter, since elements that were before discarded due to their out of bound access in the input photo, can now be multiplied with elements of the padding and then summed up to the final result, with no impact to it due to their value of zero.

The new code for the padding implementation is show in Listings 5, 6. In these codes, we have eliminated the main if statement and we are now using the new image width and height for imageW and imageH respectively. Because of that, we change the thread indexing by increasing both x and y coordinates with a factor of filterR, ensuring that each thread will access their assigned output element correctly.

```
__global__ void convolutionRowGPU(float *d_Dst, float *d_Src, float *d_Filter,
                        int imageW, int imageH, int filterR) {
    int tx = (blockIdx.x * blockDim.x) + threadIdx.x;
    int ty = (blockIdx.y * blockDim.y) + threadIdx.y;
    float sum=0;

    int xnew = tx + filterR;
    int ynew = ty + filterR;

    for (int k = -filterR; k <= filterR; k++) {
        int d = xnew + k;
        sum += d_Src[ynew * imageW + d] * d_Filter[filterR - k];
    }

    d_Dst[ynew * imageW + xnew] = sum;
}
```

Listing 5: CUDA Implementation for Row Convolution using Padding

```
__global__ void convolutionColumnGPU(float *d_Dst, float *d_Src, float *d_Filter,
                                     int imageW, int imageH, int filterR) {
    int tx = (blockIdx.x * blockDim.x) + threadIdx.x;
    int ty = (blockIdx.y * blockDim.y) + threadIdx.y;
    float sum=0;

    int xnew = tx + filterR;
    int ynew = ty + filterR;

    for (int k = -filterR; k <= filterR; k++) {
        int d = ynew + k;
        sum += d_Src[d * imageW + xnew] * d_Filter[filterR - k];
    }

    d_Dst[ynew * imageW + xnew] = sum;
}
```

Listing 6: CUDA Implementation for Column Convolution using Padding

2.7.2 Evaluation

In this section, we compare the performance of the new implementation with the previous ones. Since no change was made to the precision chosen here (floats) we won't compare the precision of this algorithm with the previous ones.

Figure 7 shows all the execution times for all the implementations. Here, we observe that the padding implementation had a significant speedup in the case of the CPU execution. This occurs because in the initial implementation, every iteration of the innermost loop, checks whether we are within valid bounds where a convolution can be performed. This adds conditional branching, which can slow down the CPU due to branch mispredictions. The new implementation gets rid of the boundary checks and thus with the use of padding ensures that the computation only occurs within the valid ranges, avoiding redundant evaluations.

As of the CUDA implementation, even though we solved the divergence problem within the WARPs, there was no performance gain between the new implementation and the previous one. The main reason this happens is because of the additional memory overhead added by the padding. In this implementation, threads accessing padded regions trigger additional global memory transactions, further exacerbating the bottleneck created by global memory access.

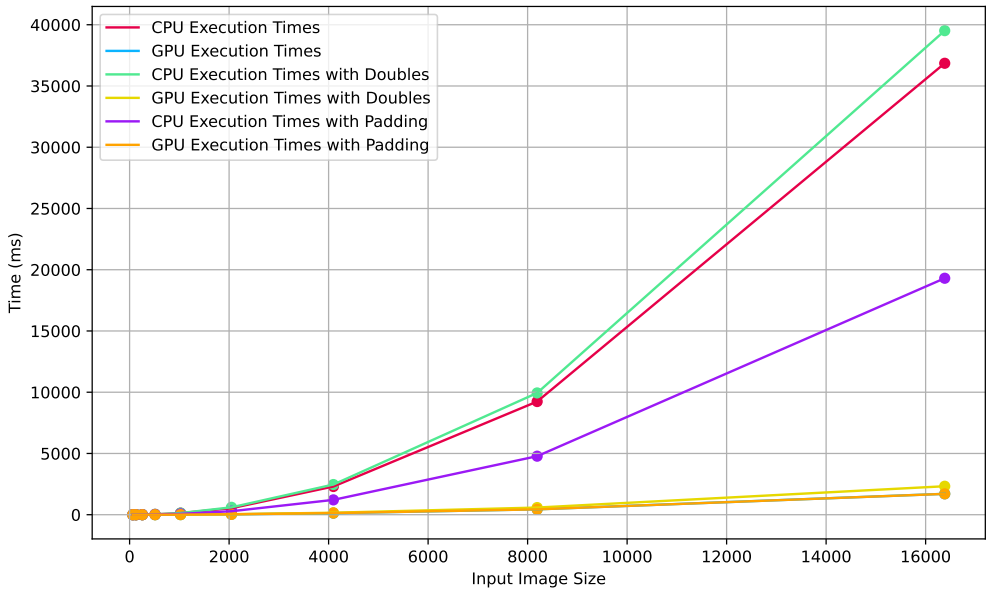

Figure 7: Execution Time Comparison between all implementations

Table 3 shows the standard deviation for the final experiments of the padding implementation.

Table 3: Standard Deviations for CPU and GPU Execution Times

Input Size	**CPU Std Dev**	**GPU Std Dev**
64	0.02470	0.00075
128	0.01502	0.00479
256	0.13146	0.02007
512	0.14832	0.03677
1024	0.88639	0.04998
2048	1.08438	0.09592
4096	5.29401	0.44197
8192	14.67845	38.08174
16384	38.96049	66.15749

A Device Query Outputs

In the following sections we provide the output of deviceQuery; NVIDIA's sampe utility that provides detailed information about the GPU(s) on a system.

A.0.1 Mars

```
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 4 CUDA Capable device(s)

Device 0: "GeForce GTX 690"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2000 MBytes
     (2097086464 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Max Clock rate:                            1020 MHz (1.02 GHz
     )
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy
     engine(s)
```

```
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 4 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 1: "GeForce GTX 690"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2000 MBytes
     (2097086464 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Max Clock rate:                            1020 MHz (1.02 GHz
     )
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy
     engine(s)
```

```
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 5 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 2: "GeForce GTX 690"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2000 MBytes
     (2097086464 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Max Clock rate:                            1020 MHz (1.02 GHz
     )
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy
     engine(s)
```

```
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 8 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 3: "GeForce GTX 690"
  CUDA Driver Version / Runtime Version          10.2 / 10.2
  CUDA Capability Major/Minor version number:    3.0
  Total amount of global memory:                 2000 MBytes
     (2097086464 bytes)
  ( 8) Multiprocessors, (192) CUDA Cores/MP:     1536 CUDA Cores
  GPU Max Clock rate:                            1020 MHz (1.02 GHz
     )
  Memory Clock rate:                             3004 Mhz
  Memory Bus Width:                              256-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy
     engine(s)
```

```
138   Run time limit on kernels:                     No
139   Integrated GPU sharing Host Memory:            No
140   Support host page-locked memory mapping:       Yes
141   Alignment requirement for Surfaces:            Yes
142   Device has ECC support:                        Disabled
143   Device supports Unified Addressing (UVA):      Yes
144   Device supports Compute Preemption:            No
145   Supports Cooperative Kernel Launch:            No
146   Supports MultiDevice Co-op Kernel Launch:      No
147   Device PCI Domain ID / Bus ID / location ID:   0 / 9 / 0
148   Compute Mode:
149      < Default (multiple host threads can use ::cudaSetDevice()
          with device simultaneously) >
150 > Peer access from GeForce GTX 690 (GPU0) -> GeForce GTX 690 (GPU1)
      : Yes
151 > Peer access from GeForce GTX 690 (GPU0) -> GeForce GTX 690 (GPU2)
      : Yes
152 > Peer access from GeForce GTX 690 (GPU0) -> GeForce GTX 690 (GPU3)
      : Yes
153 > Peer access from GeForce GTX 690 (GPU1) -> GeForce GTX 690 (GPU0)
      : Yes
154 > Peer access from GeForce GTX 690 (GPU1) -> GeForce GTX 690 (GPU2)
      : Yes
155 > Peer access from GeForce GTX 690 (GPU1) -> GeForce GTX 690 (GPU3)
      : Yes
156 > Peer access from GeForce GTX 690 (GPU2) -> GeForce GTX 690 (GPU0)
      : Yes
157 > Peer access from GeForce GTX 690 (GPU2) -> GeForce GTX 690 (GPU1)
      : Yes
158 > Peer access from GeForce GTX 690 (GPU2) -> GeForce GTX 690 (GPU3)
      : Yes
159 > Peer access from GeForce GTX 690 (GPU3) -> GeForce GTX 690 (GPU0)
      : Yes
160 > Peer access from GeForce GTX 690 (GPU3) -> GeForce GTX 690 (GPU1)
      : Yes
161 > Peer access from GeForce GTX 690 (GPU3) -> GeForce GTX 690 (GPU2)
      : Yes
162
163 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.2, CUDA
      Runtime Version = 10.2, NumDevs = 4, Device0 = GeForce GTX 690,
      Device1 = GeForce GTX 690, Device2 = GeForce GTX 690, Device3 =
      GeForce GTX 690
164 Result = PASS
```

Listing 1: Output of deviceQuery on Venus

A.0.2 Venus

```
  ./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 4 CUDA Capable device(s)

Device 0: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
```

```
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 6 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 1: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
```

```
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            No
  Supports Cooperative Kernel Launch:            No
  Supports MultiDevice Co-op Kernel Launch:      No
  Device PCI Domain ID / Bus ID / location ID:   0 / 7 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 2: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
```

```
  Device has ECC support:                         Enabled
  Device supports Unified Addressing (UVA):       Yes
  Device supports Compute Preemption:             No
  Supports Cooperative Kernel Launch:             No
  Supports MultiDevice Co-op Kernel Launch:       No
  Device PCI Domain ID / Bus ID / location ID:   0 / 132 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice()
        with device simultaneously) >

Device 3: "Tesla K80"
  CUDA Driver Version / Runtime Version          11.4 / 11.5
  CUDA Capability Major/Minor version number:    3.7
  Total amount of global memory:                 11441 MBytes
     (11997020160 bytes)
  (13) Multiprocessors, (192) CUDA Cores/MP:     2496 CUDA Cores
  GPU Max Clock rate:                            824 MHz (0.82 GHz)
  Memory Clock rate:                             2505 Mhz
  Memory Bus Width:                              384-bit
  L2 Cache Size:                                 1572864 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(65536), 2D
     =(65536, 65536), 3D=(4096, 4096, 4096)
  Maximum Layered 1D Texture Size, (num) layers  1D=(16384), 2048
     layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(16384, 16384),
      2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535,
      65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy
     engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Enabled
```

```
143   Device supports Unified Addressing (UVA):      Yes
144   Device supports Compute Preemption:            No
145   Supports Cooperative Kernel Launch:            No
146   Supports MultiDevice Co-op Kernel Launch:      No
147   Device PCI Domain ID / Bus ID / location ID:   0 / 133 / 0
148   Compute Mode:
149      < Default (multiple host threads can use ::cudaSetDevice()
          with device simultaneously) >
150 > Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU1) : Yes
151 > Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU2) : No
152 > Peer access from Tesla K80 (GPU0) -> Tesla K80 (GPU3) : No
153 > Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU0) : Yes
154 > Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU2) : No
155 > Peer access from Tesla K80 (GPU1) -> Tesla K80 (GPU3) : No
156 > Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU0) : No
157 > Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU1) : No
158 > Peer access from Tesla K80 (GPU2) -> Tesla K80 (GPU3) : Yes
159 > Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU0) : No
160 > Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU1) : No
161 > Peer access from Tesla K80 (GPU3) -> Tesla K80 (GPU2) : Yes
162
163 deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 11.4, CUDA
      Runtime Version = 11.5, NumDevs = 4, Device0 = Tesla K80,
    Device1 = Tesla K80, Device2 = Tesla K80, Device3 = Tesla K80
164 Result = PASS
```

Listing 2: Output of deviceQuery on Venus