GPGPU Programming

Donato D'Ambrosio

Department of Mathematics and Computer Science
Cubo 30B, University of Calabria, Rende 87036, Italy
mailto: donato.dambrosio@unical.it
homepage: `http://www.mat.unical.it/~donato`

Academic Year 2020/21

Table of contents

Convolution, aka Stencil Computation

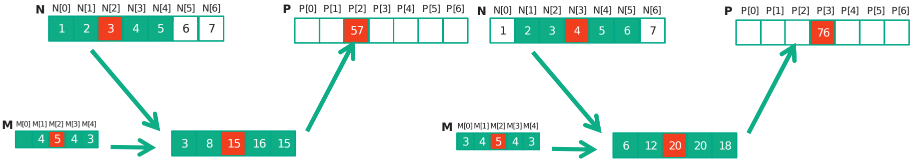
Convolution

Convolution, aka Stencil Computation

- Convolution is the basis of a wide range of parallel algorithms: signal processing, digital recording, image processing, video processing, and computer vision.
- In these areas, convolution is used as a filter that transforms signals and pixels.
- In High-Performance Computing, e.g., in numerical computation, convolution is often referred as stencil computation.
- A pros is that each data element can be calculated independently of each other.
- The cons is that there is some overlapping in the input with possible issue at boundaries.

Background

- Convolution is an array operation where each output data element is a weighted sum of a collection of neighboring input elements.
- The weights are defined by a mask array, commonly known as convolutional kernel, aka convolutional mask, which is usually invariant in time and space. Here is a 1D example with mask M and weighted sum (inner product-like) computation.

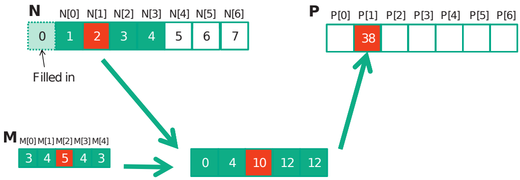



```
P[2] = N[0]*M[0]+ N[1]*M[1] + N[2]*M[2] + N[3]*M[3] + N[4]*M[4] = 57

P[3] = N[1]*M[0]+ N[2]*M[1] + N[3]*M[2] + N[4]*M[3] + N[5]*M[4] = 76
```

Background

- Because convolution is defined in terms of neighboring elements, boundary condition arise for the elements that are close to the end of the array.
- A typical approach to handling such boundary condition is to define a default value to these missing N elements. For most applications, the value is 0, since it is the null element of the product operation.

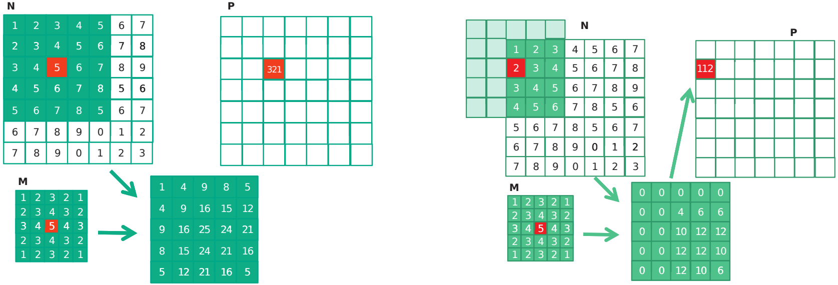



```
P[1] = 0 * M[0] + N[0]*M[1] + N[1]*M[2] + N[2]*M[3] + N[3]*M[4] = 38
```

- In this way, the missing elements, aka ghost cells, give a null contribution.

Background

- Obviously, the same issue can arise in the case of 2D convolution.



1D Parallel Convolution - A Basic Algorithm

- Let us assume that the convolution kernel (i.e., the convolution mask) is an odd number and the convolution is symmetric: `Mask_Width` is `2*n + 1` where `n` is an integer.

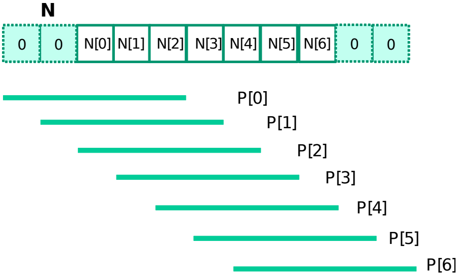
```
__global__
void convolution_1D_basic_kernel(float *N, float *M, float *P, int
    Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;

  float Pvalue = 0;
  int N_start_point = i - (Mask_Width/2);
  for (int j = 0; j < Mask_Width; j++) {
     if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
     }
   }
   P[i] = Pvalue;
}
```

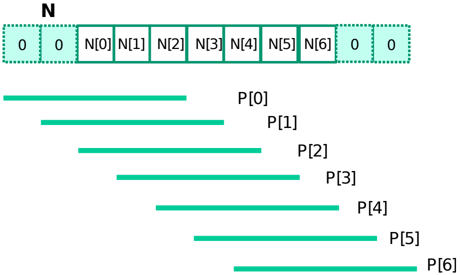
1D Parallel Convolution - A Basic Algorithm

- Let us computer the arithmetic to memory access ratio.



- For blocks with no ghost cells, each thread accesses Mask_Width elements of N, for a total of blockDim.x*(2n+1) accesses.
  - If Mask_Width is 5 and block size 1024, there are 5120 accesses per block.
- Blocks with ghost cells (the first two end the last two, in the example) do not perform mem accesses for the ghost cells.

1D Parallel Convolution - A Basic Algorithm



- The leftmost ghost cell is used by one thread; the second by two.
- In general, the number of threads that use each ghost cell (avoiding accessing any N element), from left to right, is 1, 2, ..., n, whose sum is n(n+1)/2 (Gauss formula). The same from right to left, for a total of n*(n+1) avoided accesses.
  - If n is 2, the accesses avoided are only 6.
- External blocks thus load blockDim.x*(2n+1)-n*(n+1) elements of N.

1D Parallel Convolution - A Basic Algorithm

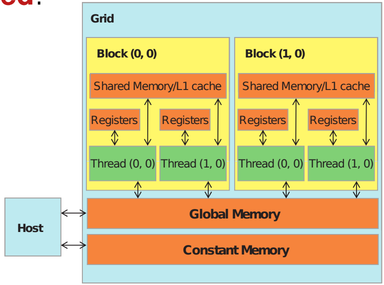
- Note that the kernel is affected by control flow divergence.

```
  //...
  for (int j = 0; j < Mask_Width; j++) {
     if (N_start_point + j >= 0 && N_start_point + j < Width) {
      Pvalue += N[N_start_point + j]*M[j];
  //...
}
```

- The cost will depend on `Width` (the size of the input array) and `Mask_Width` (the size of the mask).
  - For large input arrays and small masks, the control divergence only occurs in a small portion of the output elements, which will keep the effect of control divergence small.
  - Since convolution is often applied to large images and spatial data, we typically expect that the effect of convergence to be modest or insignificant.
- The real problem is memory bandwidth. The ratio of floating-point calculation to global memory accesses is about 1.0, meaning that we can not reach high performance.

Constant Memory and Caching

- There are three interesting properties of the way the mask array M is used that make the mask array an excellent candidate for constant memory and caching.
  - The size of the M array is typically small.
  - The contents of M never changes.
  - All threads access the M elements in the same order (for loop).
- Constant memory is like the global memory but is: **64KB** wide, read-only, **cached**!



Constant Memory and Caching

- To declare an `M` array in constant memory, the host declares it as a global variable (i.e., outside of any function) as:

```
#define MAX_MASK_WIDTH 10
__constant__ float M[MAX_MASK_WIDTH];
```

- If `M_h` stores the mask in the host memory, it can be transferred to M in the device constant memory as follows:

```
cudaMemcpyToSymbol(M, M_h, Mask_Width*sizeof(float));
```

where M is the destination, M_h the source, followed by the size.

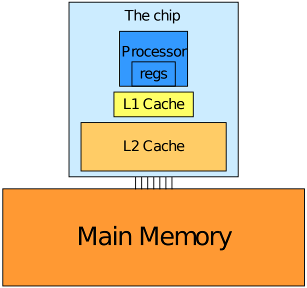
- The kernel access M as an object in the global scope[1]. The pointer to M must no longer be passed to the kernel!

```
__global__ void convolution_1D_basic_kernel(float *N, float *P,
    int Mask_Width, int Width) {
  // The body remains the same...
}
```

[1] If the kernel is implemented in a different file, it must include the external declaration of M, otherwise M is not visible to the kernel.

Constant Memory and Caching

- Modern processors commonly employ on-chip cache memories to reduce the number accesses to the DRAM. We find multiples levels of caches with different speed and size.



  - L1 caches are fast as the processor but small, e.g. 16-64KB.
  - L2 caches are slower (tens of cycles to access) but larger, e.g. 128KB-1MB, and are shared among processor cores, or SMs.
  - In some high-end processors, there are L3 caches that can be several MB in size.

Constant Memory and Caching

- A major design issue with caches is cache coherence, which arises when more cores modify L1 cached data.
- GPUs do not support cache coherence (hardware resources are all devoted to increase the arithmetic throughput).
- Nevertheless, since constant memory objects are constant, they do not present the cache coherence issue and can therefore be cached in L1!
- Furthermore, GPU caches are optimized to broadcast a value to a large number of threads: when all threads in a warp access the same constant memory variable, as is the case of M, it is delivered with high bandwidth and almost immediately.
- Since M is typically small, we can assume that all M elements are accessed from caches! The ratio of floating-point arithmetic to memory access is therefore increased from 1 to 2.

Tiled 1D Convolution with Halo Cells

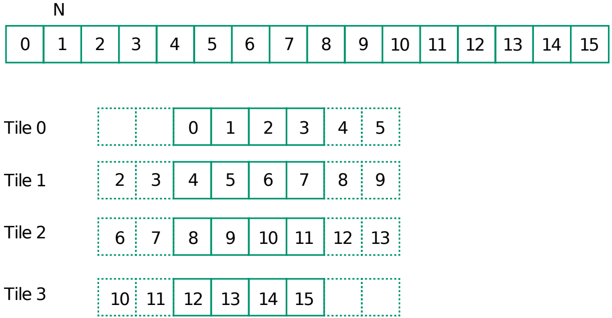
- We will now address the memory bandwidth issue in accessing N array element with a tiled convolution algorithm.
- We assume that each thread calculates one output P element.
- Let us consider a small example of 16-element 1D convolution computed using four thread blocks of four threads each (in practice, there should be 32 threads per block).

N

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Tile 0			0	1	2	3	4	5
Tile 1	2	3	4	5	6	7	8	9
Tile 2	6	7	8	9	10	11	12	13
Tile 3	10	11	12	13	14	15		

Tiled 1D Convolution with Halo Cells



- A tiled algorithm would load all input elements for calculating all output elements into the Shared Memory per each block.
- Let us assume that the mask size is an odd number equal to $2 * n + 1$. The figure highlights the required input cells if $n = 2$.
- The elements that are involved in multiple tiles are commonly referred to as halo cells or *skirt cells*.

Tiled 1D Convolution with Halo Cells

	N														
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Tile 0			0	1	2	3	4	5
Tile 1	2	3	4	5	6	7	8	9
Tile 2	6	7	8	9	10	11	12	13
Tile 3	10	11	12	13	14	15		

- We refer tiles with ghost cells (tiles 0 and 3) as boundary tiles, the others as internal tiles.
- Note that each halo belongs to two tiles (e.g., the left halo of Tile 1 also belong to Tile 0) and and therefore it is loaded twice (i.e., once per block).

Tiled 1D Convolution with Halo Cells

```
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int
    Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  __shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
  int n = Mask_Width/2;

  int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x >= blockDim.x - n) {
    N_ds[threadIdx.x - (blockDim.x - n)] =
    (halo_index_left < 0) ? 0 : N[halo_index_left];
  }
  N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

  int halo_index_right = (blockIdx.x + 1)*blockDim.x + threadIdx.x;
  if (threadIdx.x < n) {
    N_ds[n + blockDim.x + threadIdx.x] =
    (halo_index_right >= Width) ? 0 : N[halo_index_right];
  }
  __syncthreads();
  // continue to the next slide...
```

Tiled 1D Convolution with Halo Cells

```
  //...

  float Pvalue = 0;
  for(int j = 0; j < Mask_Width; j++) {
    Pvalue += N_ds[threadIdx.x + j]*M[j];
  }
  P[i] = Pvalue;
}
```

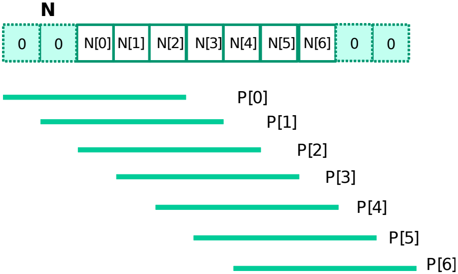
- The tiled 1D convolution kernel is significantly longer and more complex than the basic kernel.
- We introduced the additional complexity in order to reduce the number of DRAM accesses for the N elements for improving the arithmetic to memory access ratio (and thus the aritmetic intensity) so that the achieved performance is not limited or less limited by the DRAM bandwidth.
- Let us computer the arithmetic to memory access ratio...

Tiled 1D Convolution with Halo Cells

- With respect to the basic kernel, all the memory accesses correspond to the loads of the N elements into the shared mem.
- Each N element is only loaded by one thread. However, 2n halo cells will also be loaded, n from the left and n from the right, for blocks that do not handle ghost cells. Therefore, we have the blockDim.x+2n elements loaded by the internal thread blocks and blockDim+n elements loaded by boundary thread blocks.
- For internal blocks, the ratio of memory accesses between the basic and the tiled 1D convolution kernel is: $(blockDim.x * (2n + 1))/(blockDim.x + 2n)$.
- For external blocks, the ratio is: $\big(blockDim.x * (2n + 1) - n(n + 1)/2\big)/(blockDim.x + n)$.
- For most situations, blockDim.x is much larger than n; both n and n*(n+1)/2 can be neglected, and the ratios can be become: $(blockDim.x * (2n + 1))/(blockDim.x) = 2n + 1 =$ Mask_Width.

Tiled 1D Convolution with Halo Cells

- This should be quite an intuitive result. In the original algorithm, each N element is redundantly loaded by approximately Mask_Width threads. For example, N[2] is loaded by the 5 threads that calculate P[0], P[1], P[2], P[3], and P[4]. That is, the ratio of memory access reduction is approximately proportional to the mask size.

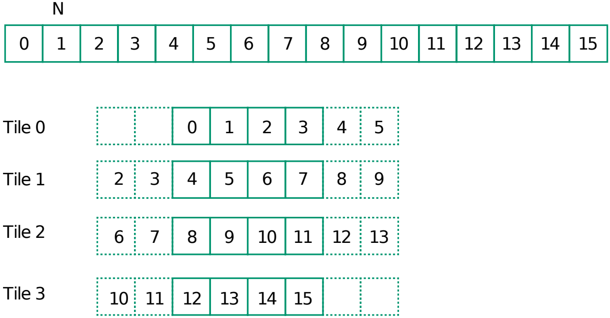


Tiled 1D Convolution with Halo Cells

- Note that, for small values of n, the n and n(2+1) terms can not be ignored.
- For example, if blockDim is 32 and n is 5 the ratio for all the internal blocks become: approximate ratio = (32*11-10)/(32+10)=8.14, while the non-approximated one is 11.
- Using small block and tile sizes may result in significantly less reduction in memory accessesthan expected.

A Simpler Tiled 1D Convolution — General Caching

- Recent GPUs provide general L1 and L2 caches, where L1 is private to each SM and L2 is shared among all SMs.
- Due to the CUDA blocks scheduling policy, there is a high probability that the halo cells of a given block are available in the L2 cache since they were accessed few moments before by the previous block as internal cells.



A Simpler Tiled 1D Convolution — General Caching

- Therefore, we can leave the accesses to these halo cells in the original N elements rather than loading them into the N_ds, that will be used to load the internal cells only!

```
__global__ void convolution_1D_tiled_kernel(float *N, float *P, int
    Mask_Width, int Width) {

  int i = blockIdx.x*blockDim.x + threadIdx.x;
  //__shared__ float  N_ds[TILE_SIZE + MAX_MASK_WIDTH - 1];
  __shared__ float  N_ds[TILE_SIZE];
  int n = Mask_Width/2;

  //int halo_index_left = (blockIdx.x - 1)*blockDim.x + threadIdx.x;
  //if (threadIdx.x >= blockDim.x - n) {
  //  N_ds[threadIdx.x - (blockDim.x - n)] =
  //   (halo_index_left < 0) ? 0 : N[halo_index_left];
  //}
  //N_ds[n + threadIdx.x] = N[blockIdx.x*blockDim.x + threadIdx.x];

  //continue...
```

A Simpler Tiled 1D Convolution — General Caching

```
//if (threadIdx.x < n) {
//  N_ds[n + blockDim.x + threadIdx.x] =
//  (halo_index_right >= Width) ? 0 : N[halo_index_right];
//}

N_ds[threadIdx.x] = N[blockIdx.x*blockDim.x+threadIdx.x];

__syncthreads();

//float Pvalue = 0;
//for(int j = 0; j < Mask_Width; j++) {
//  Pvalue += N_ds[threadIdx.x + j]*M[j];
//}

int This_tile_start_point = blockIdx.x * blockDim.x;
int Next_tile_start_point = (blockIdx.x + 1) * blockDim.x;
int N_start_point = i - (Mask_Width/2);

//continue...
```

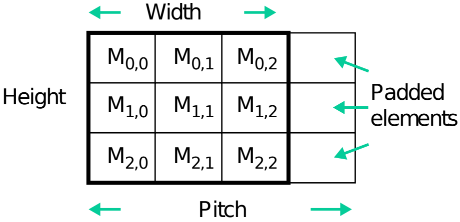
A Simpler Tiled 1D Convolution — General Caching

```
  float Pvalue = 0;
  for (int j = 0; j < Mask_Width; j++) {
    int N_index = N_start_point + j;
    if (N_index >= 0  && N_index < Width)
      if ((N_index >= This_tile_start_point)
        && (N_index < Next_tile_start_point))
        Pvalue += N_ds[threadIdx.x+j-(Mask_Width/2)]*M[j];
      else
        Pvalue += N[N_index] * M[j];
  }
  P[i] = Pvalue;
}
```

- `This_tile_start_point` and `Next_tile_start_point` are the starting index of the current tile and that processed by the next block.
- The if statement tests the current access.
    - If the `N_index` is internal for the current block, the element is accessed from the `N\_ds` array in the shared memory; Otherwise, it is accessed from the `N` array, which is hopefully in the L2 cache.

Tiled 2D Convolution With Halo Cells

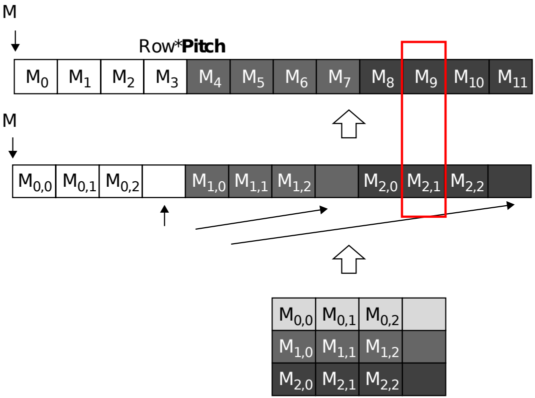
- We refer a common class of padded image format.
- Padding is a technique used to let image rows to be a multiple (in terms of bytes) of the DRAM burst size.



- Without padding, M(1,0) in row 1 would reside in one DRAM burst unit whereas M(1,1) and M(1,2) would reside in the next DRAM burst unit. Accessing row 1 would require two DRAM bursts and wasting half of the memory bandwidth.

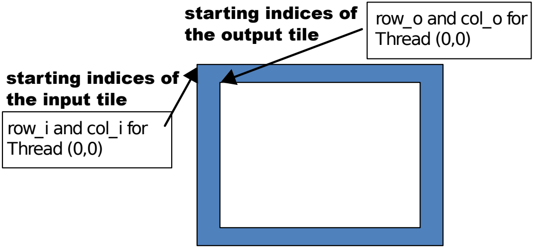
Tiled 2D Convolution With Halo Cells

- The linearized 1D index of the pixel elements will use the pitch (i.e., the row size padded elements included) instead of the width:
  `Linearized 1D index = row * pitch + column`



Tiled 2D Convolution With Halo Cells

- We are now ready to work on the design of a tiled 2D convolution kernel. We refer to the case of a single channel image.
- We need to first design the input and output tiles to be processed by each thread block, as shown in the figure.



- Note that the input tiles include the halo cells and extend beyond their corresponding output tiles by the number of halo cells in each direction.

Tiled 2D Convolution With Halo Cells

```
__global__ void convolution_2D_tiled_kernel(float *P, float *N, int
    height, int width, int pitch, int channels, int Mask_Width, const
    float __restrict__ *M)
{
  int tx = threadIdx.x; int ty = threadIdx.y;
  int row_o = blockIdx.y*O_TILE_WIDTH + ty;
  int row_i = row_o - Mask_Width/2;
  int col_i = col_o - Mask_Width/2;
  int col_o = blockIdx.x*O_TILE_WIDTH + tx;

  // All the threads participate in loading the input tiles into the
  // shared memory. Each thread check if the y and x indices of its
  // input tile elements are within the valid range of the input.
  // If not, the input element it is attempting to load is actually
  // a ghost element and a 0.0 value is placed into the shared memory.
  __shared__ float N_ds[TILE_SIZE+MAX_MASK_WIDTH-1][TILE_SIZE+
    MAX_MASK_HEIGHT-1];
  if((row_i >= 0)&&(row_i < height) && (col_i >= 0)&&(col_i < width))
    N_ds[ty][tx] = data[row_i * pitch + col_i];
  else
    N_ds[ty][tx] = 0.0f;
  // continue...
```

Tiled 2D Convolution With Halo Cells

```
  float output = 0.0f;
  // Only the threads whose indices are both smaller than
  // the O_TILE_WIDTH must compute
  if(ty < O_TILE_WIDTH && tx < O_TILE_WIDTH)
  {
    for(i = 0; i < MASK_WIDTH; i++)
      for(j = 0; j < MASK_WIDTH; j++)
        output += M[i][j] * N_ds[i+ty][j+tx];

    if(row_o < height && col_o < width)
      data[row_o*width + col_o] = output;
  }
}
```

- In a basic kernel, every thread in a thread block will perform `Mask_Width`^{2} accesses to the image array, for a total of `Mask_Width`^{2} `* O_TILE_WIDTH`^{2} accesses per block.
- In the tiled kernel, a block collectively load one input tile, resulting in `(O_TILE_WIDTH + Mask_Width-1)`^{2} accesses.

Tiled 2D Convolution With Halo Cells

- The ratio of image array accesses between the basic and the tiled 2D convolution kernel is: `Mask_Width`^{2} $*$ `O_TILE_WIDTH`^{2} `/ (O_TILE_WIDTH + Mask_Width-1)`^{2}
- The trend of the image array access reduction ratio as we vary `O_TILE_WIDTH`, the output tile size.

TILE_WIDTH	8	16	32	64
Reduction Mask_Width =5	11.1	16	19.7	22.1
Reduction Mask_Width =9	20.3	36	51.8	64

- As `O_TILE_WIDTH` becomes very large, mask size becomes negligible compared to tile size. Thus, each input element loaded will be used about (Mask_Width)^2 times. For `Mask_Width`= 5, we expect that the ratio will approach 25 as the `O_TILE_WIDTH` becomes much larger than 5. For example, for `O_TILE_WIDTH`=64, the ratio is 22.1.

Tiled 2D Convolution With Halo Cells

- This means we need large `O_TILE_WIDTH`, even if a large amount of shared memory would be needed to hold the input tiles.

TilLE_WIDTH	8	16	32	64
Reduction Mask_Width =5	11.1	16	19.7	22.1
Reduction Mask_Width =9	20.3	36	51.8	64

- For a larger Mask_Width, such as 9 in the bottom row, the best result with a tile width of 64 is 64, which is far from the ideal 81.
- Note that `O_TILE_WIDTH`=64 and `Mask_Width`=9 result into input tile of 20,736 bytes (assuming single precision data), which exceeds the amount of shared memory in SM of current GPUs[2].

[2]Stencil computation that is derived from finite difference methods for solving differential equation often require a Mask_Width of 9 or above to achieve numerical stability. Such stencil computation can benefit from larger amount of shared memory in future generations of GPUs.