# Pipeline performance in reverse kernel

**Contents**



## Goals of this tutorial

1. Learn how to address global memory in non linear way (scatter/gather operations)
2. Learn how to measure kernel performance with PyCUDA
3. Understanding how to improve memory access bandwidth using increasing pipeline load

## Reverse memory script

There is a script canvas for copy and reverse data from a source array to a destination array :

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule

# Kernel definition and on-line compilation
mod = SourceModule("""
__global__ void reverse(float *destination, float *source, unsigned int size)
{
  const int global_i = blockIdx.x*blockDim.x +threadIdx.x;
  if ( size > global_i)
  {
    // TODO: find the way to reverse data order in destination array
    destination[global_i] = source[global_i];
  }
}
""")

reverse = mod.get_function("reverse")

size_data=100

# number of threads in a block
block_size=32
# number of blocks of thread
num_blocks=(size_data+block_size-1)/block_size

source = numpy.arange(0,size_data,1).astype(numpy.float32)
destination = numpy.zeros_like(source)
execution_time= reverse(
        drv.Out(destination), drv.In(source), numpy.uint32(size_data),
        block=(block_size,1,1), grid=(num_blocks,1), time_kernel=True)
# CPU "compute" same result in CPU and display the difference with GPU computing result
print(destination - source[::-1])
```

### Fix the script

Save this script as *tp_reverse.py*.

Actually, if run this script, you can view that this script doesn't work!

run inside **ipython** interactive console:

```
andradgu@soroban:~/tp_cuda$ ipython
```

```
Python 2.7.2+ (default, Jul 20 2012, 22:15:08)
Type "copyright", "credits" or "license" for more information.

IPython 0.10.2 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object'. ?object also works, ?? prints more.

In [1]: run tp_reverse.py
[-99. -97. -95. -93. -91. -89. -87. -85. -83. -81. -79. -77. -75. -73. -71.
 -69. -67. -65. -63. -61. -59. -57. -55. -53. -51. -49. -47. -45. -43. -41.
 -39. -37. -35. -33. -31. -29. -27. -25. -23. -21. -19. -17. -15. -13. -11.
  -9.  -7.  -5.  -3.  -1.   1.   3.   5.   7.   9.  11.  13.  15.  17.  19.
  21.  23.  25.  27.  29.  31.  33.  35.  37.  39.  41.  43.  45.  47.  49.
  51.  53.  55.  57.  59.  61.  63.  65.  67.  69.  71.  73.  75.  77.  79.
  81.  83.  85.  87.  89.  91.  93.  95.  97.  99.]

In [2]: source
Out[2]:
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
        11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,
        22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,
        33.,  34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,
        44.,  45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,
        55.,  56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,
        66.,  67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,
        77.,  78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,
        88.,  89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,
        99.], dtype=float32)

In [3]: destination
Out[3]:
array([  0.,   1.,   2.,   3.,   4.,   5.,   6.,   7.,   8.,   9.,  10.,
        11.,  12.,  13.,  14.,  15.,  16.,  17.,  18.,  19.,  20.,  21.,
        22.,  23.,  24.,  25.,  26.,  27.,  28.,  29.,  30.,  31.,  32.,
        33.,  34.,  35.,  36.,  37.,  38.,  39.,  40.,  41.,  42.,  43.,
        44.,  45.,  46.,  47.,  48.,  49.,  50.,  51.,  52.,  53.,  54.,
        55.,  56.,  57.,  58.,  59.,  60.,  61.,  62.,  63.,  64.,  65.,
        66.,  67.,  68.,  69.,  70.,  71.,  72.,  73.,  74.,  75.,  76.,
        77.,  78.,  79.,  80.,  81.,  82.,  83.,  84.,  85.,  86.,  87.,
        88.,  89.,  90.,  91.,  92.,  93.,  94.,  95.,  96.,  97.,  98.,
        99.], dtype=float32)

In [4]: source[::-1]
Out[4]:
array([ 99.,  98.,  97.,  96.,  95.,  94.,  93.,  92.,  91.,  90.,  89.,
        88.,  87.,  86.,  85.,  84.,  83.,  82.,  81.,  80.,  79.,  78.,
        77.,  76.,  75.,  74.,  73.,  72.,  71.,  70.,  69.,  68.,  67.,
        66.,  65.,  64.,  63.,  62.,  61.,  60.,  59.,  58.,  57.,  56.,
        55.,  54.,  53.,  52.,  51.,  50.,  49.,  48.,  47.,  46.,  45.,
        44.,  43.,  42.,  41.,  40.,  39.,  38.,  37.,  36.,  35.,  34.,
        33.,  32.,  31.,  30.,  29.,  28.,  27.,  26.,  25.,  24.,  23.,
        22.,  21.,  20.,  19.,  18.,  17.,  16.,  15.,  14.,  13.,  12.,
        11.,  10.,   9.,   8.,   7.,   6.,   5.,   4.,   3.,   2.,   1.,
         0.], dtype=float32)
```

*source* will be reverse in destination. The right result should be identical as *source[::-1]*.

**The first step for you is to fix this script**
Add another constant int *global_o* index of destination data array

## Examine performance

Kernel in script has a extra parameter **time_kernel=True** that allows to get execution time of kernel without data transfer. Now you can try to print this execution time in function of data size with:

```
#bandwith in Mb/S (float32 take 4 bytes) x2 : 1 read & 1 write access per data element
bandwidth= 2*(size_data*4)/(execution_time*1000000)
print ("for size=%10d execution time=%12g s bandwidth=%10g Mb/s" %(size_data,execution_time,bandwidth))
```

Compute ratio between data access by seconds (bandwidth) help us to understand performance of GPU.

### Get limits of Devices

Before test to change values in data size or block and grid size, please execute *deviceQuery* that comes with CUDA SDK:

```
$ /opt/cudaSDK/C/bin/linux/release/deviceQuery
```

This program shows you limits of GPU memory, number of threads per block, grid maximum size, etc.

### Performance Vs. Data Size Vs. Block Size

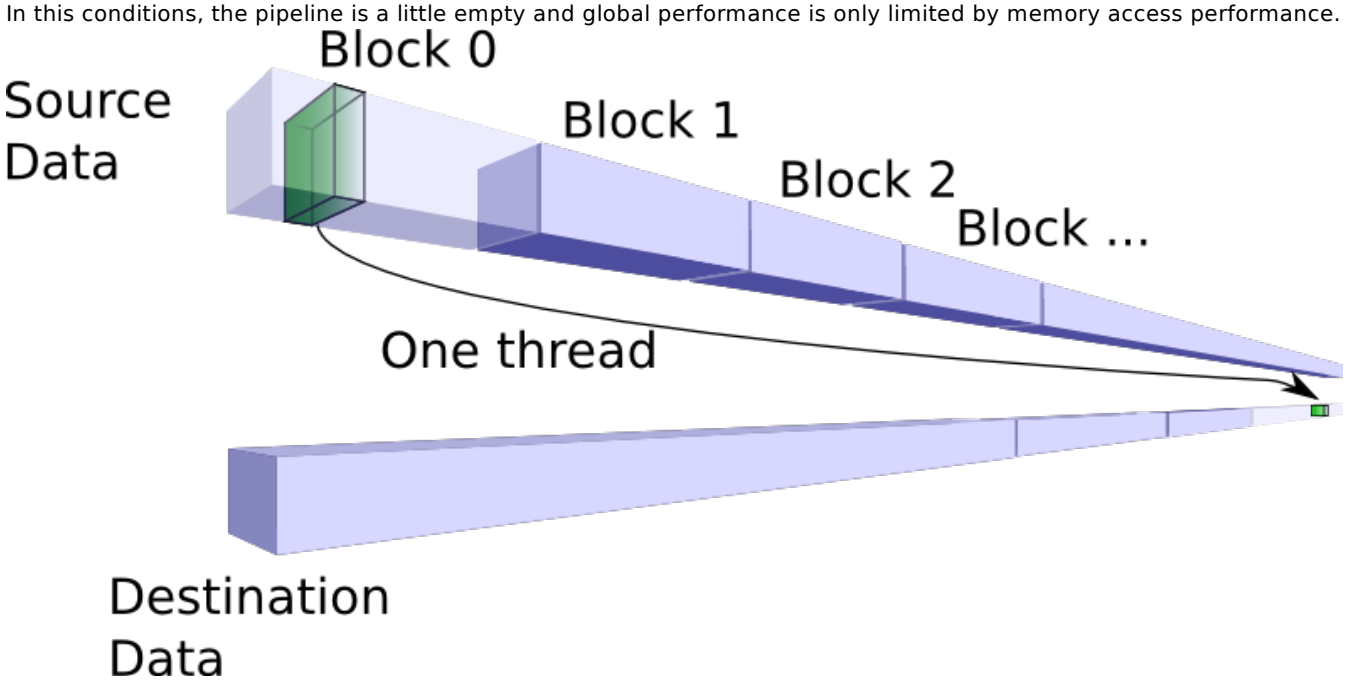
Try to change data size and note if performance is better when size of data increase, them change the size of block of threads. In some case error will occur if your kernel exceeds devices limits. Don't worry, CUDA seem to be robust to continue executions. Please, take in consideration that maybe you aren't alone: Others persons works with the device simultaneously.
For automatic test, you can insert a for block in your python script like this:

```
for size_data in [100,1000,10000, ...] :
  # number of threads in a block
  block_size=32
  # number of blocks of thread
  num_blocks=(size_data+block_size-1)/block_size
  ...
  ...
```

**What conclusion you can do ?**

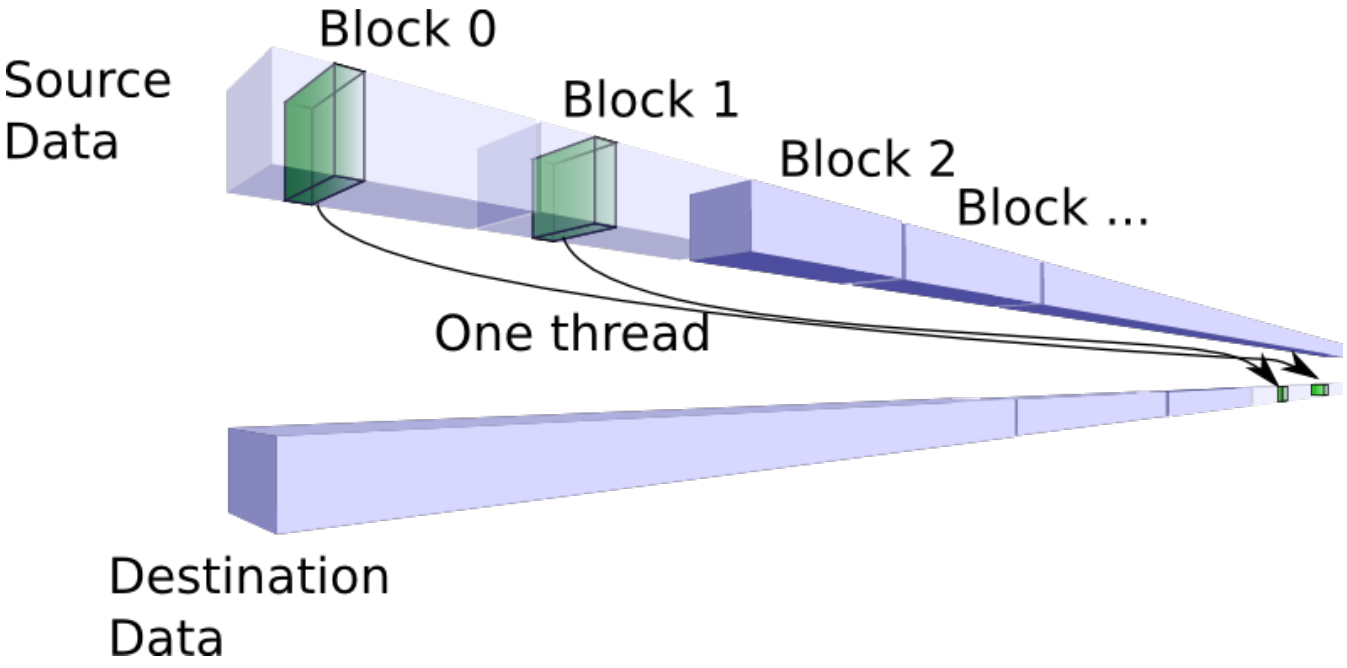
## Fill the Pipeline

Every thread of this kernel takes one data element in source array and process it to produce one element in destination array. The computation complexity are very low (only 2 or 3 additions) per data element.
In this conditions, the pipeline is a little empty and global performance is only limited by memory access performance.




### What happen if we try to fill the pipeline by processing more data per threads ?

To do that, for instance, every thread will be process a data element from a block of data but it will also process another element in other block of data




- **Modify the kernel to process 2 data elements from 2 different blocks of data in a single thread**

Instead of :

```
const int global_i = blockIdx.x*blockDim.x +threadIdx.x;
```

adding two index :

```
const int global_i_0 = (blockIdx.x*2)*blockDim.x +threadIdx.x;
const int global_i_1 = (blockIdx.x*2+1)*blockDim.x +threadIdx.x;
...
```

- Because every block of thread process the double of data as before, **you need to adjust the number of blocks in grid or size of block**.
- **In the same way, you can modify kernel to process 4 data elements per threads** them observe performance evolution.

## Automatic Code generation

- Optionally you can try to automatic generate kernel code using python string processing function:

```
kernel_code = """
bla;
bla;
""" + '\n'.join([ "const int global_i_" + str(idx)
                  + " = (blockIdx.x*4 +" +str(idx) +")*blockDim.x +threadIdx.x;"
                  for idx in range(0,4)]) + """
bla;
bla;
"""
```

That produce with **print(kernel_code)**:

```
bla;
bla;
const int global_i_0 = (blockIdx.x*4 +0)*blockDim.x +threadIdx.x;
const int global_i_1 = (blockIdx.x*4 +1)*blockDim.x +threadIdx.x;
const int global_i_2 = (blockIdx.x*4 +2)*blockDim.x +threadIdx.x;
```

```
const int global_i_3 = (blockIdx.x*4 +3)*blockDim.x +threadIdx.x;
bla;
bla;
```

Back to Parallel Computing tutorials