Gentle Introduction to GPU Programming in Numba

September 15, 2018

1 A Gentle Introduction to GPU Programming using Numba

By Lokesh Kumar T

Sept 2018

IIT Madras

2 Overview of the Talk

- CPUs and GPUs
- Whats Heterogenous Computing?
- Introduction to GPU Programming Terminologies
- Introduction to Numba
- Lets multiply a vector by 2 in GPU!
- Matrix Multiplication in a GPU
- How to proceed further?

CPUs are generally increasingly good in reducing *latency* for single stream of processing.

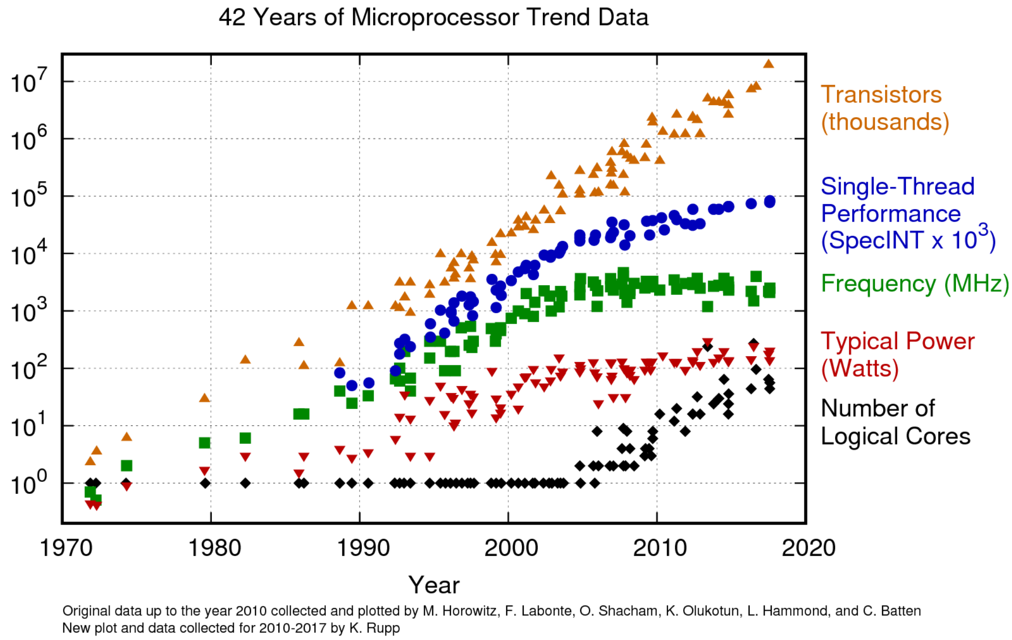
Fundamental performance quest in single core CPU:

How to devote transistors on a chip to make a single stream of instructions run faster and faster.

- **CPU**: work on a variety of different calculations
- **GPU**: best at focusing all the computing abilities on a specific task
- **CPU**: few cores (up to 24) optimized for sequential serial processing
- **GPU**: thousands of smaller and more efficient cores for a massively parallel architecture
- **GPU**s provide **superior processing power**, **memory bandwidth** and **efficiency** over their CPU counterparts.

3 Why is can't we go faster?

- Power Management
- Memory Access Rates
- Instruction Level Parallelism



trend

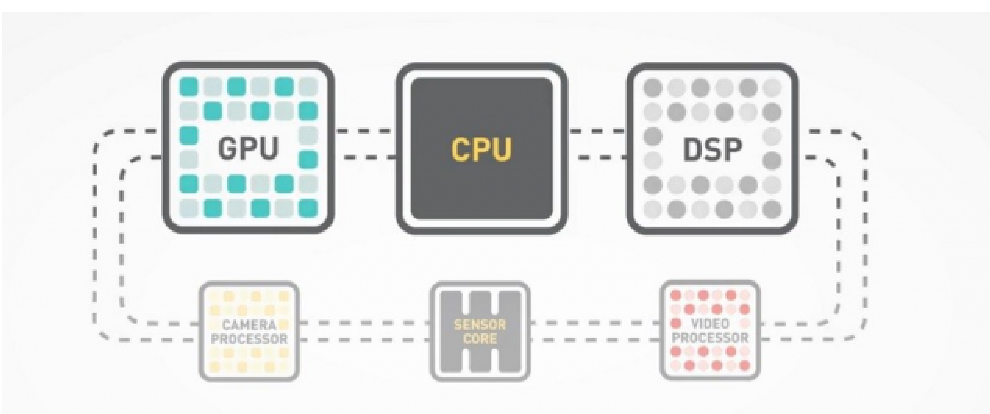
4 How to exploit the Sequential Acceleration of CPU and Parallel Acceleration of GPU?

5 Heterogenous Computing

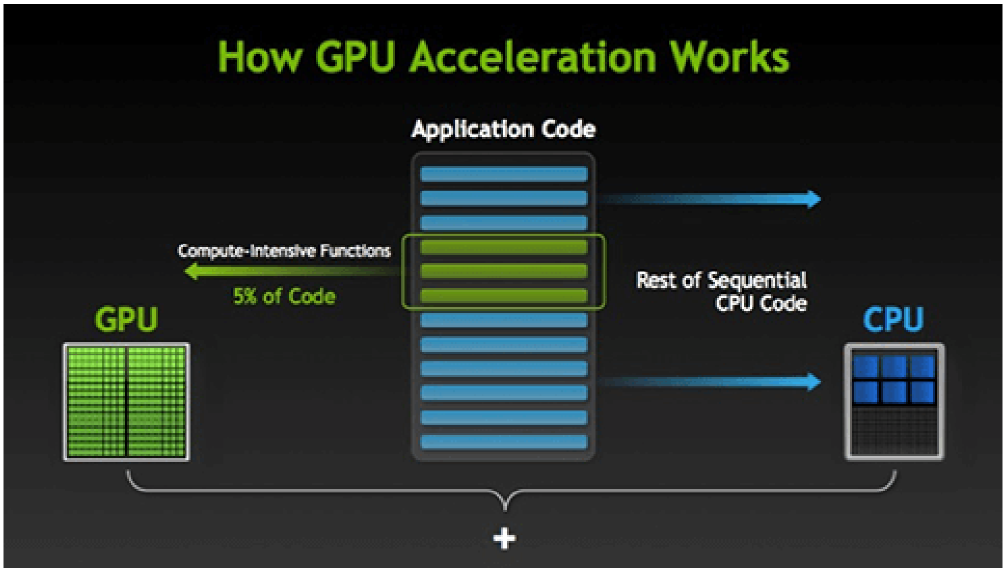
6 Basic Idea

6.0.1 How does GPU manage to accelerate compute intensive tasks?

- Uses parallel programming strategy
  - Breaks the tasks into several smaller sub tasks



heterogenous



basic_idea

- Many versions of the sub-tasks operating different data
- Works on the sub tasks simultaneously (parallely).

6.0.2 How does CPU approach this same task?

- Uses single thread. Single stream of instructions.
- Tries to accelerate that single stream of instruction.
- Sequential Processing!

6.1 What are those `tasks`?

6.2 Computational tasks

- Matrix Multiplications
- Vector Addition
- Fast Fourier Transforms

- Signal Processing techniques
- Deep Learning Workloads

6.3 Breaking a task into sub-tasks

- Crutial to attain maximum performance
- Depends from task to task
- Some tasks are easier and straight-forward than others
- Lets see an example

6.4 Vector Addition

- Vectors are columns of numbers

$$\vec{a} = \begin{bmatrix} 1 \\ 2 \\ \vdots \\ n \end{bmatrix}_{n \times 1}$$

Lets take 2 vectors $\vec{a}, \vec{b}$ both in $\mathbb{R}^n$ (n-dimensional space).

$$\vec{a} = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} \tag{1}$$

$$\vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{2}$$

Whats this quantity?

$$\vec{a} + \vec{b}$$

- Element wise addition

$$\vec{a} + \vec{b} = \begin{bmatrix} a_1 + b_1 \\ a_2 + b_2 \\ a_3 + b_3 \\ \vdots \\ a_n + b_n \end{bmatrix} \tag{3}$$

```
In [ ]: for i in range(n):
            c[i] = a[i] + b[i]
```

7 How to split it into sub tasks?

In other words parallelize it?

7.1 Here are the steps (simple algorithms):

- Identify independent instructions (operations)
- Identify their input of these indepedent operations
- Finalze your fundamental unit that will have several versions running parallely.

7.2 Revisiting Visiting Vector Addition

- Whats the fundamental operation performed?
- Whats the input for this operation to be performed?

Fundamental Operation: **Addition of 2 numbers**
Input: One Elements from 2 vectors `a[i], b[i]`
Same operation is performed on different data items. (here $a[i], b[i]$)

SIMD Processing - **S**ingle **I**nstruction **M**ultiple **D**ata Processing

7.2.1 Our Approach to program this addition in GPU:

- Replicate `addition` on different compute units in GPUs
- Give appropriate inputs to these units so that they perform useful operation.
- Aggregate the each units output and send it to CPU

7.3 Terminologies

- **Device**: GPU (Device memory: GPU Memory)
- **Host**: CPU (Host Memory: CPU Memory)
- **Kernel**: The function that runs in GPU
  - Whats our kernel in vector addition?
- **Threads**: The computational units in GPUs. Runs a version of the kernel.
- **Blocks**: Collections of a set of threads
- **Grid**: Collection of set of blocks

7.4 Lets Code Vector Scaling in GPU using Numba!

8 Sample Introduction to Numba

Numba gives you the power to speed up your applications with high performance functions written directly in Python.

We will look into a basic program and understand the Numba programming basics.

```
In [1]: # !conda install -c numba numba
        import numba
```

```
In [2]: from numba import cuda
        print(cuda.gpus)
```

```
<Managed Device 0>
```

```
In [17]: import numpy as np
         # SCALING A VECTOR BY 2
         # Create the data array - usually initialized some other way
         data = np.ones(256*4096) # 1,041,664

         threadsperblock = 256

         # Ceil function
         blockspergrid = (data.size + (threadsperblock - 1)) // threadsperblock
         print ("Blocks in one grid:\t" + str(blockspergrid))
         print ("Threads in one Block:\t" + str(threadsperblock))
```

```
Blocks in one grid:         4096
Threads in one Block:        256
```

9 Whats this threadsperblock, blockspergrid business?

- For effective parallelization of higher dimensional data structures, loopy data structures:
  - CUDA follows an hierarchy
  - threads, blocks, grids we saw remember?

9.1 Hierarchy

- **Threads**: The computational units in GPUs. Runs a version of the kernel.
- **Blocks**: Collections of a set of threads
- **Grid**: Collection of set of blocks

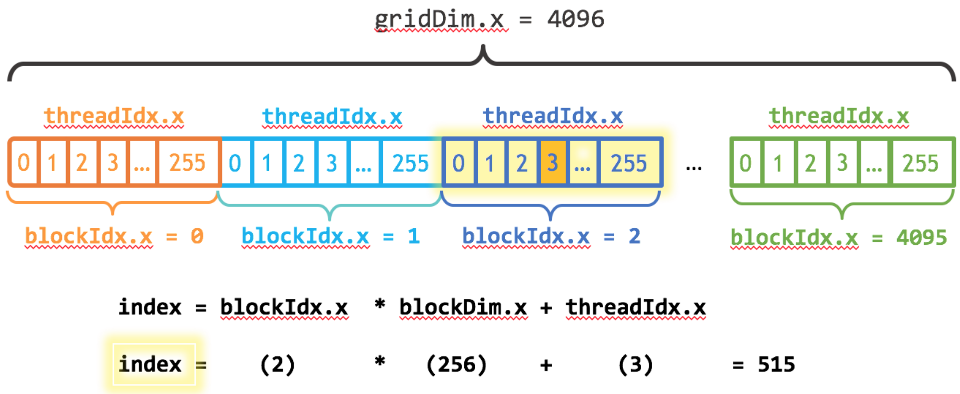
9.1.1 We defined how many blocks and threads are needed.

9.1.2 Now lets define the kernel function

```
In [18]: from numba import cuda

         @cuda.jit
         def my_kernel(io_array):
             pos = cuda.grid(1)
             if pos < io_array.size:  # Check array boundaries
                 io_array[pos] *= 2 # do the computation
```

10 Finding the global index of the thread

- **numba.cuda.grid(ndim)** - Return the absolute position of the current thread in the entire grid of blocks.



1D_blocks

10.1 Calling the Kernel from the code

```
In [19]: %%timeit
         # Now start the kernel
         # And time the GPU execution time also
         my_kernel[blockspergrid, threadsperblock](data)
```

```
13.6 ms ± 66.5 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [24]: print(data)
```

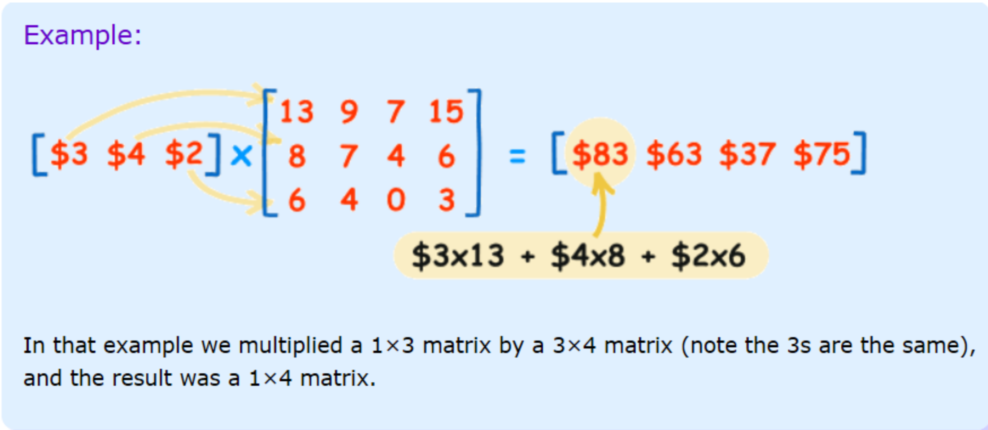
```
[2. 2. 2. ... 2. 2. 2.]
```

```
In [7]: %%timeit
        # timing the CPU Operations
        data_2 = data*2
```

```
2.35 ms ± 278 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
In [21]: @cuda.jit
         def my_kernel2(io_array):
             pos = cuda.grid(1)
             if pos < io_array.size:
                 io_array[pos] *= 2 # do the computation
```

```
In [22]: # Host code
         data = np.ones(256*4096)
         threadsperblock = 256
         blockspergrid = np.ceil(data.shape[0] / threadsperblock).astype('int32')
         my_kernel2[blockspergrid, threadsperblock](data)
```



matrix_mul

```
In [23]: print(data)
```

```
[2. 2. 2. ... 2. 2. 2.]
```

11 Lets do Matrix Multiplication in GPU!

How will you approach this problem????

How will you assign the threads and blocks?

11.1 Remember the guidelines:

- Identify independent instructions (operations)
- Identify their input of these indepedent operations
- Finalize your fundamental unit that will have several versions running parallely.

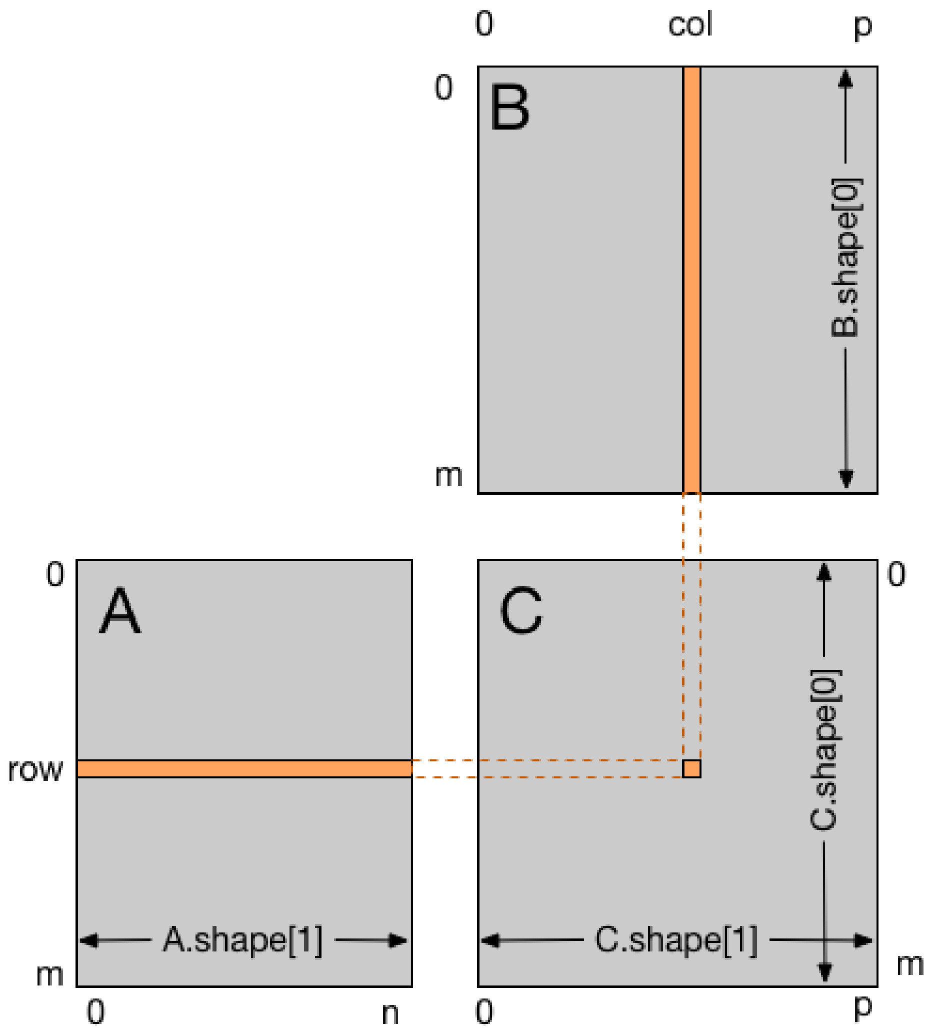
11.2 What's the dimension of the block here?

Is it 1D as we saw in scalar multiplication?

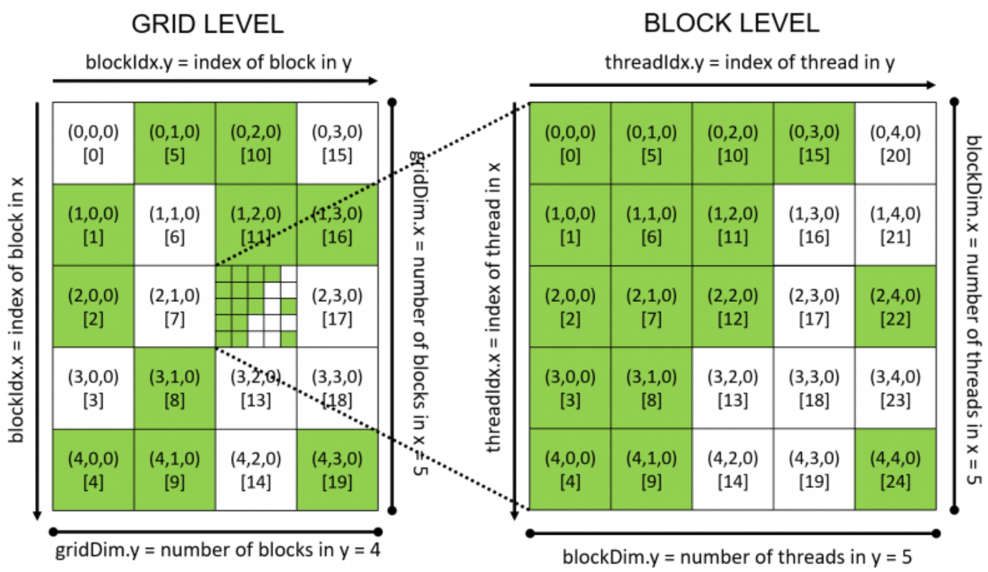
12 Lets code this in Numba

12.0.1 Host Code

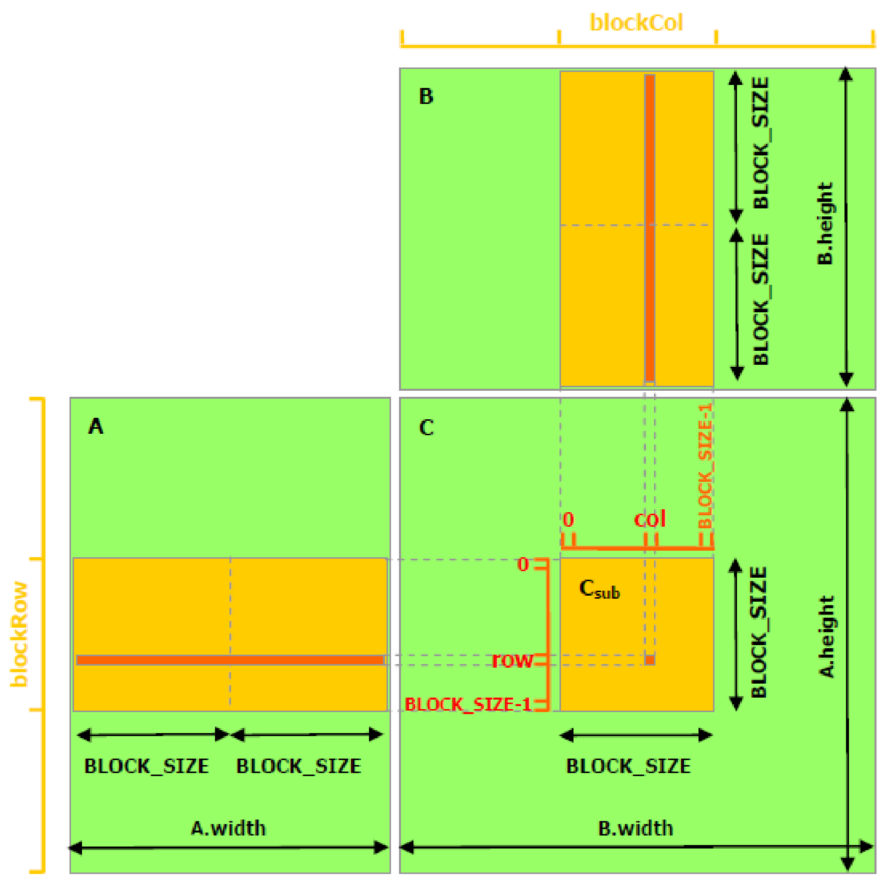
```
In [26]: # Host code

         # Initialize the data arrays
         m = 2**11 # 2048
         n = 2**11
```



matrix


GRID LEVEL
blockIdx.y = index of block in y
blockIdx.x = index of block in x
(0,0,0) [0]
(0,1,0) [5]
(0,2,0) [10]
(0,3,0) [15]
(1,0,0) [1]
(1,1,0) [6]
(1,2,0) [11]
(1,3,0) [16]
(2,0,0) [2]
(2,1,0) [7]
(2,3,0) [17]
(3,0,0) [3]
(3,1,0) [8]
(3,2,0) [13]
(3,3,0) [18]
(4,0,0) [4]
(4,1,0) [9]
(4,2,0) [14]
(4,3,0) [19]
gridDim.x = number of blocks in x = 5
gridDim.y = number of blocks in y = 4
BLOCK LEVEL
threadIdx.y = index of thread in y
threadIdx.x = index of thread in x
(0,0,0) [0]
(0,1,0) [5]
(0,2,0) [10]
(0,3,0) [15]
(0,4,0) [20]
(1,0,0) [1]
(1,1,0) [6]
(1,2,0) [11]
(1,3,0) [16]
(1,4,0) [21]
(2,0,0) [2]
(2,1,0) [7]
(2,2,0) [12]
(2,3,0) [17]
(2,4,0) [22]
(3,0,0) [3]
(3,1,0) [8]
(3,2,0) [13]
(3,3,0) [18]
(3,4,0) [23]
(4,0,0) [4]
(4,1,0) [9]
(4,2,0) [14]
(4,3,0) [19]
(4,4,0) [24]
blockDim.x = number of threads in x = 5
blockDim.y = number of threads in y = 5

image


blockCol
B
BLOCK_SIZE
BLOCK_SIZE
B.height
A
C
blockRow
0
col
BLOCK_SIZE-1
0
Csub
row
BLOCK_SIZE-1
BLOCK_SIZE
A.height
BLOCK_SIZE
BLOCK_SIZE
BLOCK_SIZE
A.width
B.width

image

```
p = 2**11

A = np.full((m, n), 1, np.float) # matrix containing all 1's
B = np.full((n, p), 1, np.float) # matrix containing all 1's
```

12.0.2 Host to device data transfer + Memory allocation in GPU

```
In [27]: # Copy the arrays to the device
         A_global_mem = cuda.to_device(A)
         B_global_mem = cuda.to_device(B)

         # Allocate memory on the device for the result
         C_global_mem = cuda.device_array((m, p))
```

12.1 Kernel

```
In [28]: @cuda.jit
         def matmul(A, B, C):
             """Perform matrix multiplication of C = A * B
             """
             i, j = cuda.grid(2)
             if i < C.shape[0] and j < C.shape[1]:
                 tmp = 0.
                 for k in range(A.shape[1]):
                     tmp += A[i, k] * B[k, j]
                 C[i, j] = tmp
```

12.2 Defining threadsperblock, blockspergrid

```
In [29]: threadsperblock = (32, 32)
         # Dimension of the matrix we defined is 2048x2048

In [30]: blockspergrid_x = int(np.ceil(A.shape[0] / threadsperblock[0]))
         blockspergrid_y = int(np.ceil(B.shape[1] / threadsperblock[1]))
         blockspergrid = (blockspergrid_x, blockspergrid_y)
```

12.3 Kernel Call

```
In [31]: # Start the kernel
         matmul[blockspergrid, threadsperblock](A_global_mem, B_global_mem, C_global_mem)

         # Copy the result back to the host
         C = C_global_mem.copy_to_host()

In [32]: print(C)

[[2048. 2048. 2048. ... 2048. 2048. 2048.]
 [2048. 2048. 2048. ... 2048. 2048. 2048.]
 [2048. 2048. 2048. ... 2048. 2048. 2048.]
```

```
 ...
 [2048. 2048. 2048. ... 2048. 2048. 2048.]
 [2048. 2048. 2048. ... 2048. 2048. 2048.]
 [2048. 2048. 2048. ... 2048. 2048. 2048.]]
```

13 Lets time it!

```
In [33]: %%timeit -n 10
         matmul[blockspergrid, threadsperblock](A_global_mem, B_global_mem, C_global_mem)
```

```
299 µs ± 134 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

```
In [34]: %%timeit -n 10
         C = A.dot(B)
```

```
463 ms ± 20.4 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

14 New Moore's Law

- Computers no longer get faster, just Wider
- Rethink your algorithms to be parallel
- Data-Parallel Computing is the most scalable solution

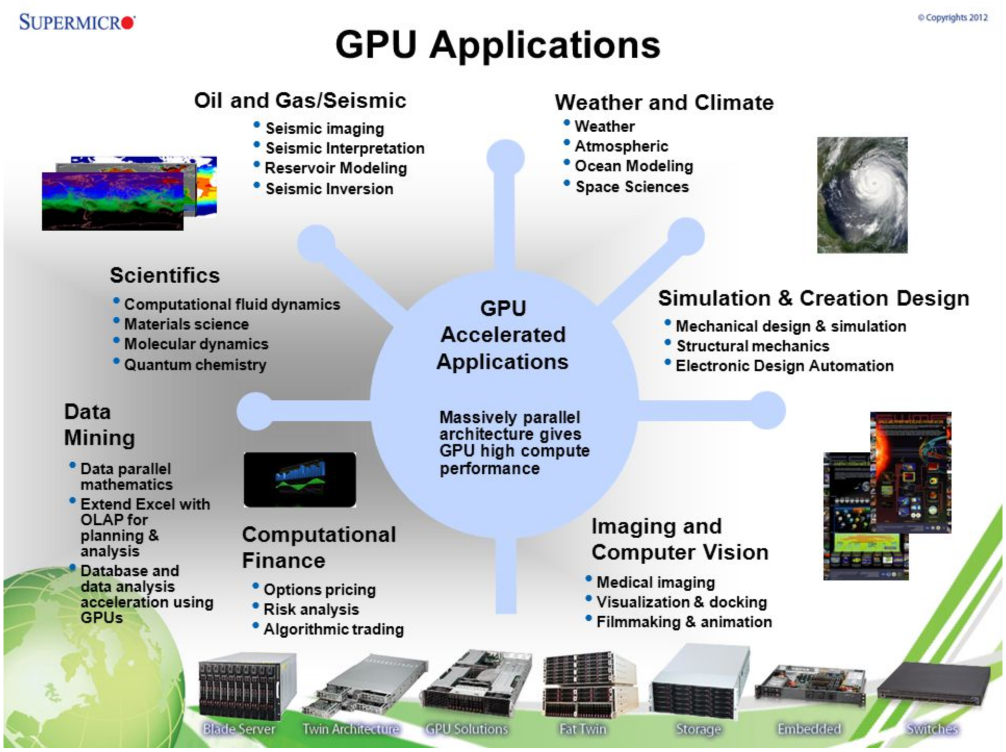
15 Summary

15.1 Thanks for your patience

This presentation and extensive resources can be found in my GitHub - tlokeshkumar.

Even projects and other codes in GPU Programming, Deep Learning etc are present in my GitHub.... Do check them out if interested!!!

Feel free to contact me at lokesh.karpagam@gmail.com



gpu-applications