GPGPU (general purpose GPU) is a set of technoligies (compiler, some libraries, GPU driver) which allows a programmer to write general purpose programs (not realted to any graphics processing), which can run on GPUs and exploit its inherent parallelism. CUDA is the example of one of thouse (and probably the most popular) GPGPU technoligies.

The way it works is the following. User writes the ordinary C program (host code) and at some point in the code he calls function, known as GPU kernel $gpuKernel(arg_1, arg_2, \ldots, arg_n)$, which will be executed on GPU. GPU kernel is programmed as if you would program a single-thread application (except the fact that you are aware that there are multiple threads, therfore you can use certain primitives to synchronize threads). This means that GPU kernel describes the instructions, that will be executed as a **thread** on a **single streaming processor(SP)**. In reality GPU have thousands of SPs. Each one of them will be executing **the same code in parallel**.

In our paper SP is responsible for evaluation of a single stack column within push/pop stacks. However each stack coulmn should do the work, specific only for this parlicular query node (split node '/a' or a leaf node '//c', etc). So how can we govern which set of instructions would be executed, depending on the query node? - by passing **configuration parameter** *param* to GPU kernel. How is that possible if we have only predefined number of arguments $(arg_1, arg_2, \ldots, arg_n)$ when we are calling GPU kernel from the host code? - we pass **array of configuration parameters** $params[i]$ instead. Each SP on the GPU has unique identifier ID. In the beginning of kernel execution each thread reads it's configuration by extracting entry from the array: $param = params[ID]$. So each thread get it's own configuration. This configuration is called **personality**. Once configuration is read from array it is saved in GPU registers (in order to save memory). A thread has exclusive access to it's registers, no one else can read information out of it.

We use the thread's configuration to adjust control flow in GPU kernel in the following way:

Algorithm 1 GPU Kernel

```
1: param = params[ID]
2: if param → isLeaf then
3:     do something
4: else
5:     if param → prefixRelation = '//' then
6:         do something else
7:     end if
8: end if
9: if param → tag = tag from open(tag) event then
10:    do something more
11: end if
12: ...
```

So this allows to perform filtering for different query nodes using a single

GPU kernel.

Because execution of each thread in the basic scenario does not depend on other threads we claim that, we have achieve intra-query parallelism. We could also have some synchronization primitives, like barrier synchronization in GPU kernel if the semantics of some operation needs to be sure that **every** thread has executed all previous instructions.

However sometimes we need not only to synchronize, but to pass information between threads. This can be done by writing this infimation either in shared memory or in global memory. Shared memory is always preferable, but it is very small. Moreover only threads from the same block can communicate through shared memory (that's why for example XPath query cannot span several blocks).