DYNAMIC PARALLELISM IN CUDA

Dynamic Parallelism in CUDA 5.0 enables a CUDA kernel to create and synchronize new nested work, using the CUDA runtime API to launch other kernels, optionally synchronize on kernel completion, perform device memory management, and create and use streams and events, all without CPU involvement.

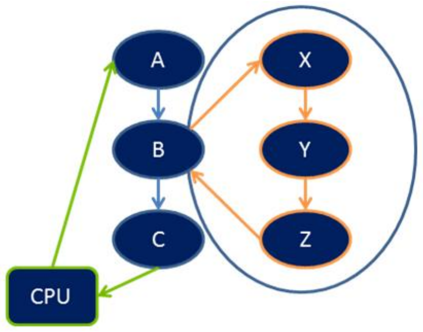
Here is an example of calling a CUDA kernel from within a kernel.

```
__global__ ChildKernel(void* data){
    //Operate on data
}

__global__ ParentKernel(void *data){
    if (threadIdx.x == 0) {
        ChildKernel<<<1, 32>>>(data);
        cudaThreadSynchronize();
    }
    __syncthreads();
    //Operate on data
}

// In Host Code
ParentKernel<<<8, 32>>>(data);
```

We call the launching kernel the "parent", and the new grid it launches the "child". Child kernels may themselves launch work, creating a "nested" execution hierarchy. Launches may continue to a depth of 24 generations, but this depth will typically be limited by available resources on the GPU. All child launches must complete in order for the parent kernel to be seen as completed. For example in the above diagram, kernel C will not be

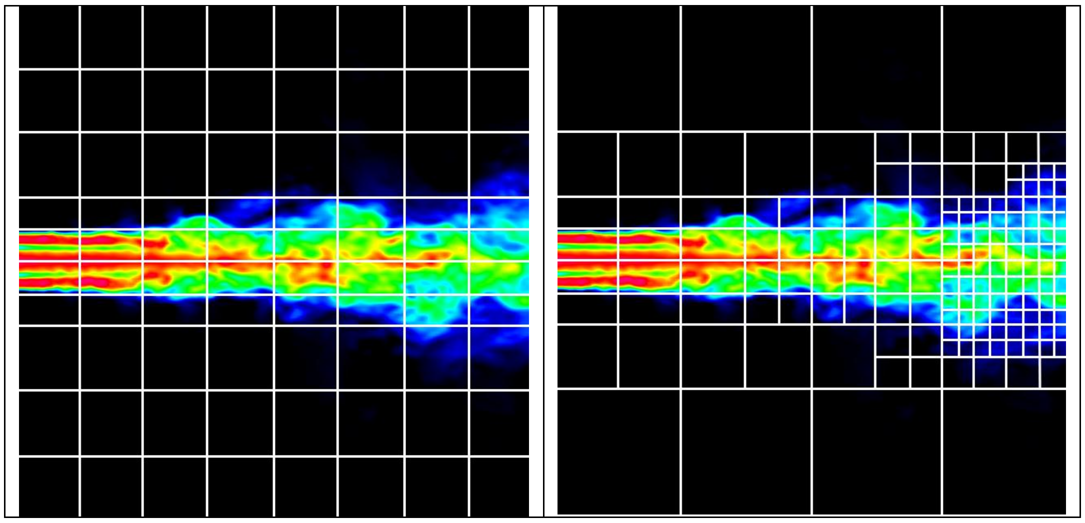


able to begin execution until kernel Z has completed, because kernels X, Y and Z are seen as part of kernel B.

The language interface and Device Runtime API available in CUDA C/C++ is a subset of the CUDA Runtime API available on the Host. The syntax and semantics of the CUDA Runtime API have been retained on the device in order to facilitate ease of code reuse for API routines that may run in either the host or device environments. A kernel can also call GPU libraries such as CUBLAS directly without needing to return to the CPU.

By using CUDA Dynamic Parallelism, algorithms and programming patterns that had previously required modifications to eliminate recursion, irregular loop structure, or other constructs that do not fit a flat, single-level of parallelism can be more transparently expressed. Program flow control can be done from within a CUDA kernel reducing PCI traffic in cases where data would otherwise have been copied back and forth between GPU and CPU between kernel launches. CUDA Dynamic Parallelism also allows for hierarchical algorithms to be written, where the data from a parent kernel computation is used to decide how to partition the next lower level of the hierarchical computation.

An example use of CUDA Dynamic Parallelism is adaptive grid generation in a computational fluid dynamics simulation, where grid resolution is focused in regions of greatest change. Without Dynamic Parallelism, performing such a simulation in CUDA requires an expensive pre-processing pass over the data.



With CUDA Dynamic Parallelism, the grid resolution can be dynamically adapted at run time based on the simulation data. Starting with a coarse grid, the simulation can “zoom in” on areas of interest and avoid unnecessary calculation in areas with little change. While this could be done using CPU launched kernels, it is more efficient for the GPU to refine the grid directly by analyzing and launching additional work as needed.