Concurrent CPU and GPU Execution using HPX and the CUDA Driver API

***Damond Howard*, (Hartmut Kaiser)**

Louisiana State University, Center for Computation and Technology, Baton Rouge, LA 70803.

STE||AR

stellar.cct.lsu.edu

LSU

Abstract

The CPU and GPU are two very different computing devices, and are meant to handle different types of computation. CPUs have fewer cores that can each handle more work per core and while the GPU has thousands of lightweight cores, making them good for smaller computations that need to be repeated often. By making use of GPUs and CPUs concurrently any applications can see performance gains. The project that will allow a developer to easily integrate GPU acceleration into an already parallel application, by making use of HPX and the CUDA driver API .The goal of this project is to make full use of a device's computational capabilities, as well as provide an easy to use abstraction that would allow it to be integrated into any existing HPX application. The library makes use of HPX components, which are classes that expose HPX actions, in order to manage GPU processes, such as launching kernels allocating memory, and loading modules, as well as handle synchronization between the CPU and GPU through the use of hpx::futures. All the user would have to do is write all the needed CUDA kernels, and create a buffer holding holding the kernel's parameters.

Introduction

The library works by creating a CUDA device component which handles context creation and memory management, a CUDA kernel component which handles loading modules containing kernels, a buffer component that holds parameters for CUDA kernels and an event component that handles synchronization for all of the components. The biggest challenge because pointers cannot be passed across nodes, would be memory management without needing to write a wrapper for each kernel.

HPX Components

HPX components are objects that expose HPX actions and can be created remotely. Components are made up of three parts and come in three different types. The parts of a component are the stubs class which handles invoking the action, the server class which contains the actual functionality of the object and the class that makes use of both the server and the stubs class and interfaces with the actual client program. The three types of components are managed,simple and template.

Example Use:

```
//create a string to hold the cuda kernel source
std::stringstream kernel;

ss << "extern \"C\" __global__ void kernel( ";
ss << def_line.str() << ", unsigned int vector_size, unsigned int
number_of_used_threads ) { \n";
ss << "\tint idx = blockDim.x * blockIdx.x + threadIdx.x; \n";
ss << "\tfor(unsigned int i = 0; i < ";
ss << "(vector_size + number_of_used_threads - 1) /
number_of_used_threads; ++i) {\n";
ss << "\t\tif(idx < vector_size) { \n";
ss << "\t\t\t" << eval_line.str() << "\n";
ss << "\t\t\tidx += number_of_used_threads;\n";
ss << "\t\t}\n";
ss << "\t}\n";
ss << "}\n\n\n\n";

 // Get list of available CUDA Devices.
 std::vector<device> devices = get_devices( hpx::find_here())

 // Check whether there are any devices
 if(devices.size() < 1)
 {
     hpx::cerr << "No CUDA devices found!" << hpx::endl;
     return hpx::finalize();
 }

     device cuda_device = devices[0]; // Create a device component
from the first device found

     buffer outbuffer = cuda_device.create_buffer(13); // Create a
buffer

      = cldevice.create_program_with_source(hello_world_src); //
Create the hello_world device program

     prog.build(); // Compile the program

     kernel hello_world_kernel = prog.create_kernel("hello_world"); //
Create hello_world kernel

     hello_world_kernel.set_arg(0, outbuffer); // Set our buffer as
argument

     hpx::cuda::work_size<1> dim;
     dim[0].offset = 0;
```

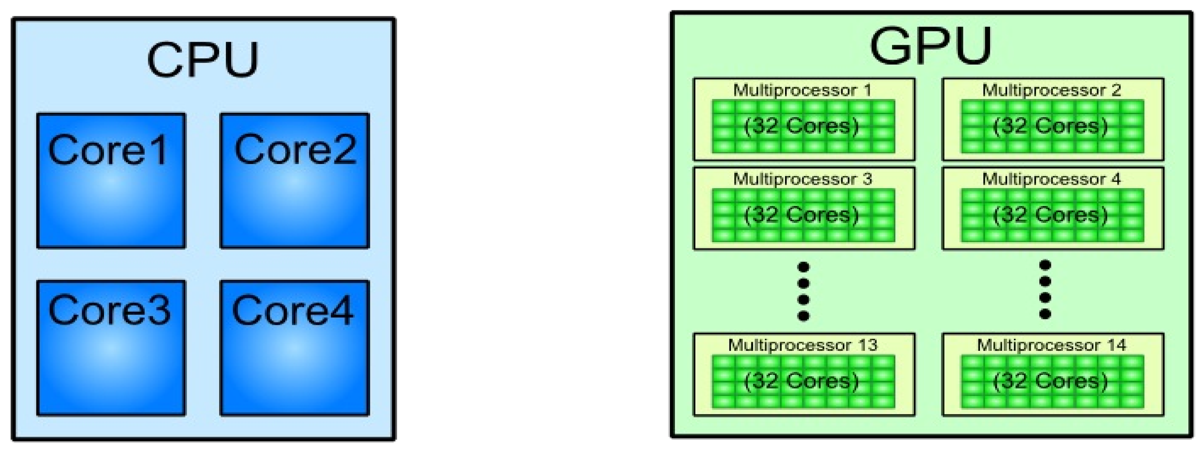
Conclusions

The goal of this project is to take advantage of the GPU as well as the CPU as efficient as possible in multi-threaded applications. As the project continues it is important to take into account the types of computation that are best handled by each device when implementing the code and also to break

CPU Vs GPU Execution

CPU	GPU
Made up of few very powerful cores	Made up of thousands of less powerful cores
Can easily hand problems that branch of and are irregular.	Is very good at handling problems in which the same operation must be repeated.
There is a certain amount of overhead for every thread launched on the CPU.	Good at launching and running 1000s of threads simultaneously.
Optimizes the speed of every individual thread so that the program runs faster over all.	Doesn't care as much about the speed of one thread but optimizes for throughput, running as many threads as possible.
Threads can run for longer periods of time	Threads do better if they don't persist for very long

CPU/GPU Architecture Comparison



Future work

In the future the library will provide easy integration of GPU acceleration into hpx applications along side the opencl version. There will be support for using multiple CUDA devices, and wrappers for all CUDA driver functions as well as error handling. Memory management must also be efficiently implemented

Acknowledgments

Louisiana State University,

Center for Computation and Technology

STE||AR group