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.

**Introduction**

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.

**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 take 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.

**Example Use:**

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

// define the kernel
kernel << "extern " #include "global void kernel()";
kernel << "int main()" << "{ unsigned int vector_size, unsigned int number_of_used_threads = 0;\nvector_size = vector_size *** \nnumber_of_used_threads = \n1 + \nvector_size *** \nnumber_of_used_threads - 1) / \nnumber_of_used_threads; \n+\nvector_size *** \number_of_used_threads *** \nvector_size = \nintiline = \n";

// get list of available CUDA devices
std::vector<device> devices = get_devices(hpx::find_device());

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

device cuda_device = devices[0]; // Create a device component from the first device found
buffer outbuffer = cuda_device.create_buffer(1024); // Create a buffer

// Load the CUDA kernel
kernel.hello_world_kernel = prog.create_kernel("hello_world");

// Set arguments for argument
outbuffer.hpx::size() = dim;
outbuffer[0].offset = 0;

// Run the kernel
hpx::cuda_work_size = 100;
```

**CPU Vs GPU Execution**

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

**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