TR-2008-04

On Using Multiple CPU Threads to Manage Multiple GPUs under CUDA

Hammad Mazhar

Simulation Based Engineering Lab
University of Wisconsin Madison

August 1, 2008

Abstract

Presented here is a short guide on how to set up a CUDA program so that it uses two different GPUs on two different cpu threads, with each GPU kernel executed from one thread. This implementation is specific to Windows XP and uses events to synchronize and time threads.

Contents

1. Introduction

This document outlines a basic framework to create multi-threaded multi-GPU CUDA programs. This is important because it allows the leveraging of multiple GPUs working with the same set of data at the same time. Data can be divided between two cards, which reduces computation time and can even double the speed of a simulation. There are many ways to implement threads in C++, this method uses Windows XP specific commands to create threads and events. Events allow threads to synchronize by making them wait for that event to occur. This makes sure that one card doesn't process data before it is ready or process data out of sync with the other card. The code provided can be expanded for additional GPUs by creating an extra thread for each card. For simplicity, this example does not provide a kernel.

2. Overview of process

The application starts execution in the main function, where two threads called Thread1 and Thread2 are created and begin executing. The main function outputs the data and waits for the two threads to finish processing. Each thread sets the device to its respective number, 0 for Thread1 and 1 for Thread 2. The threads execute the CUDA code which in turn executes the kernels that process the data. Once the kernel finishes and returns to the calling thread it activates its respective event. The main function waits for each event's activation signal; once both events are activated it resets the events and displays the new data. The main thread and two child threads loop and continue to output and process data. In this example the two child threads do not wait for an event before processing; only the main thread i.e., the master thread waits for the two child threads to finish processing.

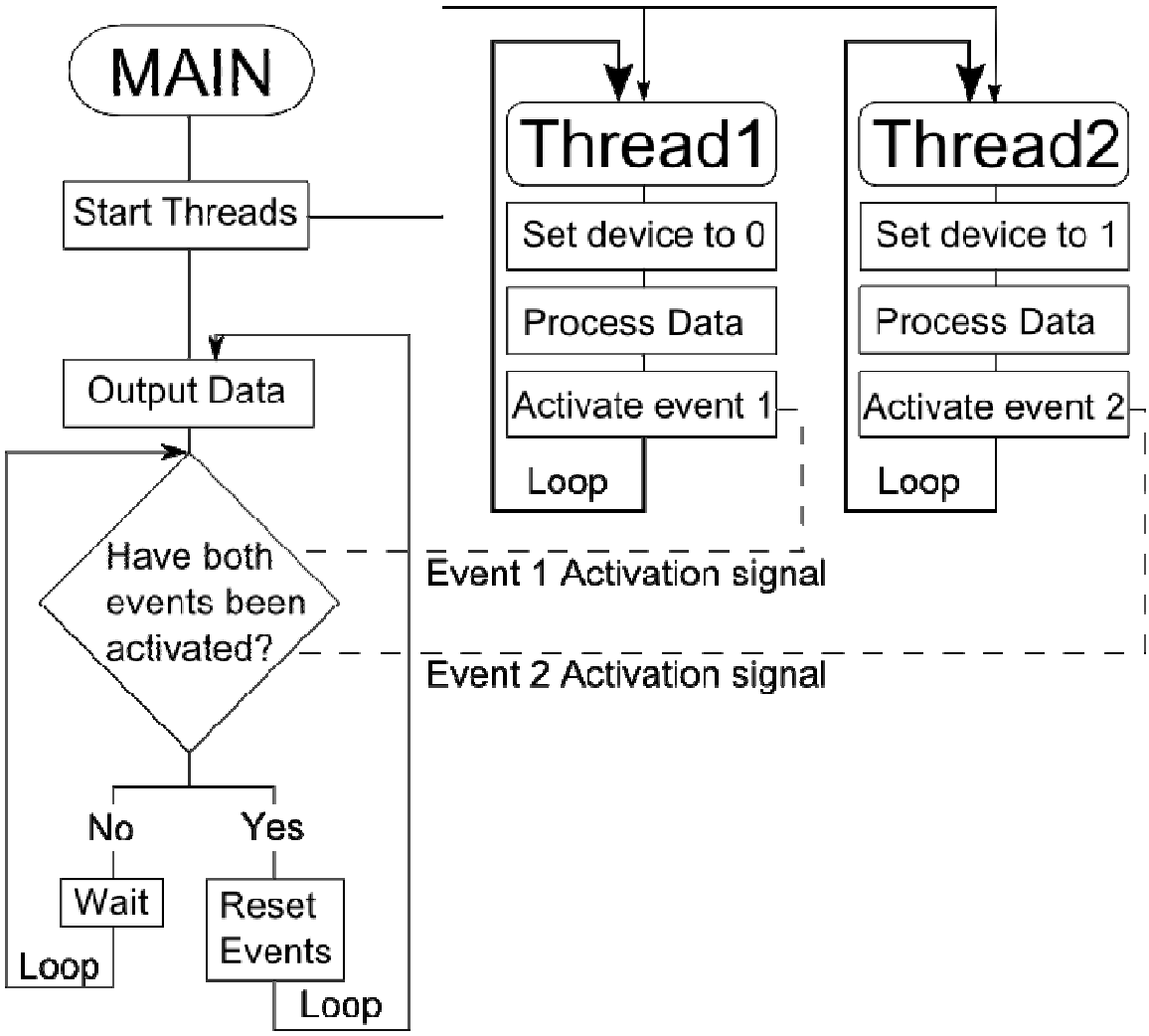

Figure 1: Application Flow. Dashed lines indicate events being activated.

3. Source

Main.cpp is the code that creates and runs the main thread as well as the two child threads, one for each device. *Simulate.cu* contains CUDA specific code for the initialization and simulation functions. `Initialize` initializes the CUDA device to the device number specified. `Simulate` copies and initializes data to the device, runs the kernel, and then copies the data back to the main memory. *Simulate_kernel.cu* contains the kernel definition that processes the data; this kernel definition is a stub and does not modify or process any data.

1.1 main.cpp

```
//main.cpp
//Includes necessary for threads and IO
#include <windows.h>
#include <process.h>
#include <stdio.h>
#include <cutil.h>
//create an array of 2 event Handles
HANDLE hEvent[2];

//Declare data variables
float pointA[32]={0};
float pointB[32]={0};

// extern "C" allows a C++ class to use Cuda code written in C
//(From the .cu file)
//.cu files cannot define/include C++ objects or classes

//Initialize switches to the device specified
extern "C" void Initialize(int device);

//Simulate runs the kernel
extern "C" void Simulate(float *dataA,float *dataB);

//First Thread Function
static void  Thread1 (LPVOID lpParam)
{
	//loop forces thread to run forever, thread does not wait but
	//sets an event that is waited upon in the main thread
	while (true)
	{
		//set active CUDA device to first device
		Initialize(0);
		Simulate(pointA,pointB);
		//Turns event on so that it can be processed
		SetEvent(hEvent[0]);
	}
}

//Second Thread Function
static void  Thread2 (LPVOID lpParam)
{
	//Like first thread function but sets second event
	while (true)
	{
		//set active CUDA device to second device
		Initialize(1);
		Simulate(pointA,pointB);
		SetEvent(hEvent[1]);
	}
}
//main function that runs main thread
void main(int argc, char **argv)
{
	//declare local variables
	int i=0;
```

```
	//create two events
	//Second Parameter specifies whether event is reset manually
	//if FALSE, event is reset after it has been waited upon
	//if TRUE,  ResetEvent(HANDLE hEvent); needs to be called
	//Third Parameter specifies the initial state when created
	hEvent[0] = CreateEvent( NULL, FALSE, TRUE, NULL );
	hEvent[1] = CreateEvent( NULL, FALSE, TRUE, NULL );

	//start two new threads
	_beginthread( Thread1, 0, NULL );
	_beginthread( Thread2, 0, NULL );

	//loop runs 10 times
	//loop prints data and then waits for new data to be calculated
	//before printing again
	while (i<10)
	{
		++i;
		//prints data
		for(int i=0; i<32; ++i){
			printf("%f %f\n",pointA[i], pointB[i]);
		}
		//----
		//use if only a single event needs to be waited upon
		//WaitForSingleObject(HANDLE hEvent, INFINITE );
		//----

		//use this if multiple threads need to be waited upon
		//2=Number of objects
		//hEvent = array of Handles for event
		//Wait for all threads to finush (true)
		//Time to wait for threads to finish (INFINITE)
		WaitForMultipleObjects(2,hEvent,TRUE,INFINITE);
	}
	//helper function that prompts user to press the “Enter” key
	//before exiting
	CUT_EXIT(argc, argv);
}
//END OF FILE
```

1.2 simulate.cu

```
//simulate.cu
#include <stdio.h>
#include <stdlib.h>
#include <cutil.h>
#include "simulate_kernel.cu"

extern "C" void Initialize(int device)
{
	//sets the current device to device, device starts at 0
	//NOTE:THIS DOES NOT CHECK TO SEE IF DEVICE IS CUDA COMPATABLE
	CUDA_SAFE_CALL(cudaSetDevice(device));
}
extern "C" void Simulate(float *dataA,float *dataB)
{
```

```
	//Local pointers to data arrays (not allocated)
	float *DevDataA=0;
	float *DevDataB=0;

	//Allocate memory for variables on device
	CUDA_SAFE_CALL(cudaMalloc((void**) &DevDataA, sizeof(float)*32));
	CUDA_SAFE_CALL(cudaMalloc((void**) &DevDataB, sizeof(float)*32));

	//copy memory from system memory to device memory
	CUDA_SAFE_CALL(cudaMemcpy(DevDataA, dataA, sizeof(float)*32,
	cudaMemcpyHostToDevice));
	CUDA_SAFE_CALL(cudaMemcpy(DevDataB, dataB, sizeof(float)*32,
	cudaMemcpyHostToDevice));

	//kernel runs one block with 32 thread
	sim_kernel<<<1,32>>>(DevDataA,DevDataB);

	//Returns string if kernel fails
	CUT_CHECK_ERROR("Kernel execution failed");

	//copy memory from device to system memory
	CUDA_SAFE_CALL( cudaMemcpy(dataA, DevDataA, sizeof(float)*32,
	cudaMemcpyDeviceToHost));
	CUDA_SAFE_CALL( cudaMemcpy(dataB, DevDataB, sizeof(float)*32,
	cudaMemcpyDeviceToHost));

	//free memory used by variables
	CUDA_SAFE_CALL(cudaFree(DevDataA));
	CUDA_SAFE_CALL(cudaFree(DevDataB));
}
//END OF FILE
```

1.3 simulate_kernel.cu

```
//simulate_kernel.cu
//definition of kernel
__global__ static void sim_kernel(float *DataA, float *DataB)
{
	//Process and modify data here
}
//END OF FILE
```