GPU Application Development Guide

Platform Symphony
Version 5.1
June 2011

Platform Computing

Contents

About this guide

This document provides development and configuration guidelines for Symphony GPU applications. It also includes tutorials for building, packaging, deploying, and running the GPU sample client and service.

Symphony GPU API Classes

All classes except the Service classes remain the same as in Symphony. The GPU specifics for Service classes are explained below.

All Service classes with the exception of the ServiceContainer class remain the same as in Symphony. To enable the application to use Symphony GPU features, use GpuServiceContainer instead of ServiceContainer,. The GpuServiceContainer inherits from ServiceContainer and extends its functionality.

Tutorial: Synchronous Symphony C++ GPU client

Goal

This tutorial guides you through the process of building, packaging, deploying, and running the GPU sample client and service. It also walks you through the GPU sample client application code.

Note:

: Contrary to standard Symphony applications, the GPU Symphony application cannot run in the DE environment since it utilizes advanced scheduling functionality that is only present in the cluster version of Symphony. Consequently, you need both the DE and Cluster versions of Symphony to complete all the steps in this tutorial.

You learn the minimum amount of code that you need to create a client.

At a glance

Before you begin, ensure you have installed the following:

1. Build environment:
   - NVIDIA SDK version 3.2 or 4.0
   - appropriate NVIDIA CUDA runtime environment
   - Platform Symphony DE 5.1
   - GPU sample application package for Symphony 5.1
2. Runtime environment
   - NVIDIA SDK version 3.2 or 4.0
   - Appropriate NVIDIA CUDA runtime environment
   - Platform Symphony 5.1
   - GPU add-on package for Symphony 5.1
   - At least one host with at least one GPU device, which is set to EXCLUSIVE_THREAD mode (1 for both CUDA 3.2 and 4.0)

In this tutorial, you will complete the following tasks:

1. Build the GPU sample client and service
2. Package the GPU sample service
3. Add the GPU application
4. Run the GPU sample client and service
5. Walk through the code

Build the GPU sample client and service

You can build client application and service samples at the same time. In your build environment:

1. Change to the `conf` directory under the directory in which you installed Symphony DE.

   For example, if you installed Symphony DE in `/opt/symphonyDE/DE51`, go to `/opt/symphonyDE/DE51/conf`.
2. Source the environment:
   - For `csh`, enter:

     ```
     source cshrc.soam
     ```
   - For `bash`, enter:

     ```
     . profile.soam
     ```
3. Go to $SOAM_HOME/5.1/samples/CPP/GpuSampleApp32 or $SOAM_HOME/5.1/samples/CPP/GpuSampleApp directory for 32 bit or 64 bit GPU applications, respectively. (For the purpose of this tutorial, we will assume usinga 64 bit GPU application). Compile using the Makefile located in the chosen directory:

   make

Package the sample service

To deploy the service, you first need to package it.

1. Change to the directory in which the compiled samples are located:

   cd $SOAM_HOME/5.1/samples/CPP/GpuSampleApp/Output/
2. Create the service package by archiving and compressing the service executable file:

   tar -cvf GpuSampleServiceCPP.tar GpuSampleServiceCPP

   gzip gpuSampleServiceCPP.tar

   You have now created your service package `GpuSampleServiceCPP.tar.gz`.

Add the application

This section will explain how to add the sample application to the cluster version of Symphony. But before you add the sample application to Symphony, you must first add a consumer that will be associated with the application. Remember to perform all these steps when you are connected to the PMC of your runtime environment, i.e., Symphony 5.1 cluster.

Create a consumer

When you add a consumer through the PMC, you must use the Add Consumer wizard. This wizard defines the properties of the consumer to be associated with your application.

1. In the PMC, click Consumers.

   The Consumers & Plans page displays.
2. Select Global Actions > Create a Consumer.

   The Create a Consumer page displays.
3. Insert GpuSampleAppCPP as a name for this consumer.
4. Add Admin as Administrator for this consumer.
5. Add Guest as User for this consumer.

6. Specify the OS username, so that the tasks of this application will be executed with its permissions.
7. Check both ComputeHosts and ManagementHosts as specified Resource Groups.
8. Review your selections, and then click Create.

   The window closes and you are now back in the Platform Management Console. Your consumer is ready to use.

Add the application

The next step is to add your application. When you add an application through the PMC, you must use the Add Application wizard. This wizard defines a consumer to associate with your application, deploys your service package, and registers your application. After completing the steps with the wizard, your application will be ready to use.

1. In the PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Select Global Actions > Add/Remove Applications.

   The Add/Remove Application page displays.
3. Select Add an application, then click Continue.

   The Adding an Application page displays.
4. Select Use existing profile and add application wizard. Click Browse and navigate to your application profile.
5. Select your application profile xml file, then click Continue.

   For GpuSampleApp, you can find your profile in the following location:

   - Windows—TBD
   - Linux—`$SOAM_HOME/5.1/samples/CPP/GpuSampleApp/GpuSampleApp.xml`

   The Service Package location window displays.
6. Browse to the service package you created in .gz or tar.gz format and select it. Click Continue.

   The Confirmation window displays.
7. Review your selections, then click Confirm.

   The window displays indicating progress. Your application is ready to use.
8. Click Close.

Run the sample client and service

In your runtime environment:

1. Change to the conf directory under the directory in which you installed Symphony Cluster.

   For example, if you installed Symphony in `/opt/ego`, go to `/opt/ego/soam/5.1/conf`.
2. Source the environment:

   For csh, enter:

   source cshrc.soam

   For bash, enter:

. profile.soam

3. Copy 'GpuSampleClient' from `$SOAM_HOME/5.1/samples/CPP/GpuSampleApp/Output` to a directory that is accessible from the runtime environment. Note that $SOAM_HOME relates to your DE environment.
4. Run 'GpuSampleClient' from the directory you copied it to. You should see output on the command line as work is submitted to the system. The client starts and the system starts the corresponding service. The client displays messages indicating that it is running.

Review and understand the samples

Review the sample client application code to learn how you can create a synchronous GPU client application.

Locate the code samples

Files	Location of Code Sample
Client	`$SOAM_HOME/5.1/samples/CPP/GpuSampleApp/SyncClient`
Message object	`$SOAM_HOME/5.1/samples/CPP/GpuSampleApp/Common`
Service code	`$SOAM_HOME/5.1/samples/CPP/GpuSampleApp/Service`
Application profile	The service required to compute the input data along with additional application parameters are defined in the application profile: `$SOAM_HOME/5.1/samples/CPP/GpuSampleApp/ GpuSampleApp.xml`
Output directory	`$SOAM_HOME/5.1/samples/CPP/GpuSampleApp/Output/`

What the sample does

The GPU client application sample opens a session and sends 10 input messages. For every input message, the client creates two vectors of 100 floats each and populates them with randomly generated numbers. The input message is sent to the GPU Service, which calculates the sum of every two vectors within a particular task using CUDA API on a GPU device scheduled by Symphony. The client application is a synchronous client that sends input and blocks the output until all the results are returned.

Note:

The sample optionally can get input parameters controlling the number of times to rerun the test, number of input messages and their lengths, and whether the service should calculate using CUDA or OpenCL API. To learn more about how to run the client with non-default options, run 'GpuSampleClient -h'.

Review the sample code

The GPU sample client application code is similar to the code sample of the basic Symphony client application. Refer to the Symphony Application Development Guide and the GPU sample client application source code to learn more about how to create GPU client applications.

Tutorial: Your first Symphony C++ GPU service

Run the sample application

Refer to 'Tutorial: Synchronous Symphony C++ client tutorial' to learn how to build, package and run a sample application.

Walk through the code

Review the sample service code to learn how you can create a service.

Locate the code samples

Refer to 'Tutorial: Synchronous Symphony C++ client tutorial' to learn about the code and executable output locations.

What the sample does

The service receives two vectors containing float numbers, their sizes, and the target calculation method. The input is considered valid only if the vectors' sizes are equal. The target calculation method can be either 'cuda' or 'opencl'. The service verifies whether it was scheduled with a valid GPU device; if so, it calculates the sum of the vectors on the scheduled GPU device using the target calculation API. If the service was not scheduked with a valid GPU device, the calculation is performed on the CPU.

Input and output:

Refer to Symphony Application Development Guide to learn more about application input and output in Symphony.

Define a service container:

For a service to be managed by Symphony, it needs to be in a container object. This is the service container. For a service to have extended GPU functionality and advanced GPU-related API, it needs to inherit from the GpuServiceContainer class:

In `SampleService.cpp`, we inherited from the ServiceContainer class.

```
#include "GpuServiceContainer.h"
class MyServiceContainer : public GpuServiceContainer
```

Process the input:

As opposed to a standard Symphony Service, the Symphony GPU Service calls GpuInvoke() on the GPU service container once per task. After you inherit from the GpuServiceContainer class, implement handlers so that the service can function properly. This is where the calculation is performed. The principles for input/output handling of the GPU service container are the same as for the standard Symphony service container.

Run the container:

Refer to the Symphony Application Development Guide to learn more about how to run the container.

Catch exceptions:

Refer to the Symphony Application Development Guide to learn more about how to catch exceptions in a Symphony service.

GPU application development

Developing GPU clients

There are no changes between the way to develop client for standard Symphony application and the way to develop GPU Symphony client. Refer to SymphonyApplication Development Guide to learn more about how to develop Symphony clients.

Developing GPU services

About GPU services

You create your service by extending GpuServiceContainer class and implementing the required handler methods. The main difference between regular Symphony service and GPU Symphony service is that the only required method is onGpuInvoke() and not onInvoke(), which is not part of the GPU service container API. All other methods such as onCreateService(), onSessionEnter() etc. remain unchanged. All principles as to Service lifecycle, Service Instance lifecycle, timeouts and optional handlers API etc are the same if you substitute onInvoke() with onGpuInvoke(). However, there is extended GPU Service Container API, which provides additional GPU related information and control options.

About GPU scheduling

The GPU scheduling is performed by Symphony on service level and takes place before onCreateService () is called. It is done by automatic selection one of the available devices and setting CUDA_VISIBLE_DEVICES accordingly. Once the service started running, all tasks associated with it, will be executed on the same device scheduled during its initialization. If the scheduling method is exclusive, the GPU device will not be available for other services until the particular service is shut down.

GPU API

Method	Description
onGpuInvoke()	Has the same meaning and signature as onInvoke() and explained above
onGpuEccError (soam::TaskContextPtr& context, int ecc)	This handler is executed by Symphony after every task. Symphony provides the number of double bit volatile ECC errors found for GPU device scheduled for particular task. This number can be either 0 if no ECC errors were found or positive number indicating the number errors found. This handler is optional. In its default implementation, Symphony fails the task and blocks the host.
getAssignedGpuDeviceId()	The method returns the ID of GPU device scheduled by Symphony to the service or -1 if the scheduling failed for some reason. This method is a helper method that can be called in any stage of GPU service lifecycle.
getLastGpuError()	The method returns the error number if the scheduling fails and getAssignedGpuDeviceId() returns -1. The possible values are GPU_SYSTEM_ERROR and GPU_INSUFFICIENT_DEVICES. This method is a helper method and should be called if getAssignedGpuDeviceId () returns -1.

Method	Description
getLastGpuErrorDetails()	The method returns the string containing additional information regarding the possible reasons for GPU scheduling failure. This method is a helper method and should be called if getAssignedGpuDeviceId() returns -1

GPU application configuration

Error handling

Error handling in addition to other application-specific behavior can be configured in the application profile. See "Service error handling control" in the Symphony documentation. Note that `onGpuInvoke()` does not currently have its own error handler in the application profile. If an error occurs in `onGpuInvoke()`, it is handled according to `onInvoke()` error settings in the application profile.

GPU scheduling and application profile

Symphony schedules one available GPU device per GPU service instance. In case of exclusive Symphony scheduling, the device is exclusively used by the service until the service is shut down. The choice of which GPU device to schedule can be controlled to a certain extent by defining specific rules and variables in the application profile of the GPU application.

Resource requirement

A basic knowledge of resource requirement strings (resreq) in Symphony is assumed. Refer to Symphony Knowledge Center to learn more about resreq.

Generally, there are two levels of scheduling in Symphony. At the global level, Symphony schedules the host where particular service instance of the application will run. At the local level, service instance initialization contains GPU device scheduling. Ideally, the service will not start on a host that does not have available GPU devices compatible with restrictions defined in the application profile. This is especially useful when not all hosts in the Resource Group assigned to your application have GPU devices.

You can control which hosts will be potential candidates for running application workload. By setting resreq correctly, you can restrict global scheduling, for example, to hosts with a valid number of GPU devices set to gpuexclusive_thread mode. To define gpuexclusive_thread mode, proceed as follows:

1. In the PMC, click Symphony Workload > Configure Applications.

   The Applications page displays.
2. Click GpuTestApp.

   The Application Profile (GpuTestApp) page displays.
3. Select Advanced Configuration in the drop-down list at the top of the page.
4. In the Resource Requirements field, replace the current contents with the following:

   select(gpuexclusive_thread > 0 && gpuexclusive_thread < 9)
5. Click Save.

   The Confirmation window displays.
6. Review your selections, and then click Confirm.

Profile environment variables

- SI_GPU_COMPUTE_MODE

  Controls the mode of the target GPU device. Only the devices that match the configured mode will be scheduled to the application's services. Configuration: 0 - for shared mode, 1 - for exclusive thread mode. Note that setting this variable to 0, will instruct Symphony to schedule only devices with shared mode set. The Symphony scheduling itself will remain exclusive as explained previously. If this variable is not set at all or set to an empty value, Symphony will not check the mode of GPU device before scheduling it to the service. Always set SI_GPU_COMPUTE_MODE to either 0 or 1 if there are prohibited GPU devices in your system.

- SI_GPU_CAPABILITY

  Controls the capability version of the target GPU device. Only the devices that match the configured GPU capability will be scheduled to the application's services. For example, the value can be 1.1 or 1.3 etc. If you decide to use this variable, you can specify that the application uses the particular resource (Symphony 'rusage'); in this case, gpucap1_1, gpucap1_3 etc. Then Symphony will know to schedule the services associated with the application according to the availability of this resource on the host. For example, if there is a host with two GPU devices, one of them with capability 1.1 and another one with capability 2.0, to restrict your application to devices with capability 1.1, follow the process described in the previous section (Resource Requirement) and update the 'Resource Requirement' with the following string:

  select(gpuexclusive_thread > 0 && gpuexclusive_thread < 9) rusage(gpucap1_1=1)

  To learn more about 'rusage' in Symphony, refer to the Symphony Knowledge Center.

Advanced configuration

Creating a GPU resource group

If not all hosts in the cluster have GPU devices, creating a Resource Group containing only GPU hosts should be considered. This method can be used instead of defining a Resource Requirement string if it is more convenient. Follow the guidelines in the 'Understanding resource groups' section inthe Symphony Knowledge Center to create tag 'cuda' and assign it to all GPU-enabled hosts in your cluster. You can then assign this Resource Group to your GPU application.