Summer Student Project Report

Konstantinos Samaras-Tsakiris

9 September 2015

Project Goal

The goal of my summer student project was the exploration of prototype systems for the integration of GPUs in the CMS offline software framework. Any proposed solution would have to address the issues of portability in the absence of GPUs, ease of use and efficiency.

1 Researching different approaches

The first part of my project consisted of researching various alternatives for integrating GPUs into the CMSSW. I researched the following approaches to various extents:

- OpenMP 4
- OpenACC
- ArrayFire
- CUDA

The prototypes can be found in this repository: https://github.com/HPC4HEP/GPUIntegration/tree/cudaWrapper

Each approach was examined in the light of the 3 main goals of portability, ease of use and efficiency. As far as portability is concerned, according to the requirements of CMSSW, the solution would have to be runtime portable, not only compile-time portable: compiled once, it should function both in machines with and without GPUs.

1.1 OpenMP 4 & OpenACC

Both of these concepts involve compiler directives for instructing the compiler to automatically offload work to the GPU. However, the current implementation of OpenMP 4 in GCC doesn't support GPUs and OpenACC is supported in GCC 5.1 only experimentally: that is, there is little flexibility yet in the thread organization and management.

Conclusion: They provide an attractive approach to parallelization, because they are much simpler to learn than CUDA and lower the effort of parallelizing existing code, but due to the above limitations I didn't experiment on them.

1.2 ArrayFire

ArrayFire is a mathematical library of many algorithms (including histograms) that are implemented with multiple backends: CPU, CUDA, OpenCL. There is a moderately rich resource of examples available online, citing good performance and ease of use. ArrayFire introduces in C++ the concept of a very powerful class called... "array", around which the entire library is built. Writing in ArrayFire feels on some accounts similar to writing Matlab. It is compile-time portable, as it can be configured to link the user's source code against either of the 3 backends, but making a runtime portable solution out of ArrayFire in the context of CMSSW would probably be quite involved. Also, it still has some rough edges, especially involving its parallel-for capability (called "gfor"), which currently lacks much functionality.

Conclusion: It is an approach that merits further study (including a benchmark comparing a native CUDA histogram with the ArrayFire implementation), as it is quite user-friendly. However the portability issue needs to be addressed.

1.3 CUDA

It is the native language of NVidia GPUs. CUDA is a C language extension that allows to execute C functions on the GPU (called kernels) and is gradually becoming more C++ compatible. As it is necessary for implementing custom GPU algorithms, CUDA is the approach I spent the rest of my project exploring. I assessed some recent advances and proceeded to build a system that integrates CUDA in CMSSW.

1.3.1 CUDA Limitations

CUDA is evolving, but traditionally its programming model is cumbersome, quite intricate in the workings of memory (as the programmer has to explicitly deal with 2 separate memory spaces interconnected over PCIe) and, most crucially, not portable. In fact, not even compilable with GCC – CUDA requires its own compiler, NVCC. This last problem is overcome by compiling CUDA code in separate compilation units and linking it to normal C++ code (normal as in GCC-compiled); however, the portability issue remains, along with the difficulty in programming. Another major issue is that CUDA is not compatible with STL data structures, but the Thrust library is bridging that gap.

1.3.2 Unified Memory

In CUDA 6.5 the concept of Unified Memory was introduced. It provides a virtually unified memory space, which eases the programming effort involved in managing 2 distinct pointers to the same data and doing explicit memory copies. This is accomplished with an automated memory management system, which migrates managed memory pages towards the device that needs them.

This system is not yet perfect; on many use cases it comes with a slight performance hit. It doesn't allow modifying the page size and under specific circumstances denies access to data that logically should have been available in that side of the bus (more information on the Issues page of the aforementioned Github repo). However, work on NVidia's part is ongoing and even hardware support has been advertised for the memory space unification (for SoCs).

1.3.3 Streams

A mechanism for task parallelism in CUDA. Apart from enabling a task-oriented programming model, streams also allow for simultaneous kernel execution and data transfer on the device, thus improving efficiency. In the latest versions of CUDA, a compiler option enables implicitly associating a different stream with each host (software) thread with low overhead.

1.3.4 Automatic kernel launch configuration

Kernel launches require 2 parameters related to how many threads will execute the function on the device: the grid and the block size. Variation of these parameters leads to great differences in kernel execution efficiency. CUDA 6.5 introduced an API to establish an optimal value for these parameters automatically using a heuristic that aims for maximum use of the hardware's resources. Work on this issue has been expanded by NVidia's Mark Harris on his Github project "HEMI".

2 Integrating CUDA in CMSSW: CudaService

Most of this project was spent on building a system that actually enables writing CUDA in CMSSW and calling kernels from GCC-compiled C++ code. This system is centered around a new CMSSW service called CudaService, which provides an asynchronous API for submitting tasks to GPUs and receiving an std::future to get the result. Its main features are:

- Launch task –> get future programming model
- Support executing fallback CPU versions of each task, if no GPU is available on the current system
- Handle common GPU failures, like temporarily insufficient resources to launch kernel, and forwards others to the user
- cudaPointer, a smart pointer class modelled after std::unique_ptr, which manages data shared between CPU and GPU

Before going any further, it is beneficial to compare traditional CUDA with my service in an illustrative example. Traditional CUDA would look like this:

```
1 float *a, *d_a;
2 a= (float*)malloc(sizeof(float));
3 cudaMalloc((void**)&d_a, sizeof(float));
4 *a= 1.0;
5 cudaMemcpy(d_a, a, sizeof(float), cudaMemcpyHostToDevice);
6 someKernel<<<grid,block>>>(d_a);
7 cudaMemcpy(a, d_a, sizeof(float), cudaMemcpyDeviceToHost);
8 cout<<*a;
9 cudaFree(d_a); free(a);
```

Using CudaService on the other hand looks like this:

```
int size, param;
cudaPointer<float> data(size);
auto GPUResult= cudaService->cudaLaunch(size, simpleTask_wrapper, param, data);
...<other work until GPUResult is needed>...
GPUResult.get();
```

2.1 System overview

The relevant source code resides in another Github repository (fork of CMSSW): https://github.com/Oblynx/cmssw/tree/cudaService/FWCore/Services

2.1.1 Kernel & Kernel Wrapper

The service's user needs to write of course the actual CUDA kernel that will be executed on the device. The kernels have to be written in a separate .cu file (automatically compiled with NVCC), in a library that will be linked to the code that calls them. However, CudaService, being native C++ code, cannot express the actual kernel launch syntax – it is NVCC-specific. For that reason, and to make supplying the (optional but recommended) CPU fallback easier, the user has to write a wrapper for each kernel, which simply calls the kernel. The "wrapper recipe" is always the same, no matter how complex the kernel is:

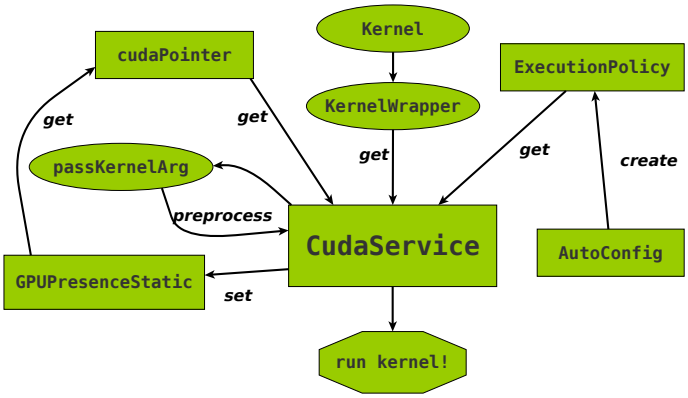

Figure 1: System architecture overview

Listing 1: Kernel wrapper recipe

```
__global__ void simpleKernel(const float* in, float* out) {...}
void simpleCPUfallback(const float* in, float* out) {...}
void simpleWrapper(bool GPUpresent, cuda::ExecutionPolicy execPol,
          const float* in, float* out){
  if(GPU) simpleKernel<<<execPol.getGridSize(), execPol.getBlockSize()>>>(in, out);
  else  simpleCPUfallback(in, out);
}
// Even if the kernel is very complex:
__global__ void complexKernel(int param1, float param2, const double* in1,
                const double* in2, double* out1, int* out2) {...}
void complexCPUfallback(int param1, float param2, const double* in1, const
            double* in2, double* out1, int* out2) {...}
void complexWrapper(bool GPUpresent, cuda::ExecutionPolicy execPol, int param1,
          float param2, ...){
  if(GPU) complexKernel<<<execPol.getGridSize(), execPol.getBlockSize()>>>(
                      param1, param2, in1, in2, out1, out2);
  else  complexCPUfallback(param1, param2, in1, in2, out1, out2);
}
```

As shown, the wrappers accept:

1. A bool argument passed by CudaService, which determines whether the kernel or the fallback should be used.
2. An object that determines the kernel launch configuration.
3. The kernel's arguments.

Then they launch the kernel and forward the arguments. The above recipe should be used for all kernels.

2.1.2 cudaPointer

This is a smart pointer class that implicitly manages the GPU memory allocation and migration of data between the CPU and the GPU, wherever they are needed. It acts like an std::unique_ptr, in that it contains a pointer of type T* (where T is an arithmetic type), that is allocated at the object's construction and freed at the object's destruction (RAII). Thus, it maintains ownership of its data throughout its lifetime. It is also portable, as it will allocate and deallocate memory using the GPU unified memory system (mentioned previously) if a GPU is available, or using the standard C++ new and delete otherwise.

Due to the way the GPU Unified Memory system works, it is illegal to access from the CPU cudaPointers that have been submitted to a kernel before getting the result of the computation. This is also the case for read-only input data, unfortunately. (see "Issues" at the prototypes repository)

There are 2 different versions of cudaPointers, modelled after std::unique_ptr:

- cudaPointer<T>: pointers to single data elements
- cudaPointer<T[]>: pointers to arrays of type T elements

The array version is a template specialization. For every kernel argument that is a pointer to arithmetic type, a cudaPointer object needs to be given when calling the kernel in its place. For example:

```
__global__ void kernel(int param, float* array) {...}
```

would be called as:

```
int param;
cudaPointer<float> array;
// <initialize param, array>...
cudaServicePtr->cudaLaunch(execPol, kernelWrapper, param, array);
```

Internally, the service takes care to only pass to the kernel wrapper the pointer to float that is contained in this example's cudaPointer<float>. When passed to the kernel and when released, cudaPointer objects change their "CUDA stream attachment", that is, their visibility scope between the host and the current stream. This also provides a mechanism to check for illegal attempts to access the cudaPointer's data from the CPU before the GPU task returns, without causing a bus error.

2.1.3 cuda::ExecutionPolicy

These are simple structs that store the kernel launch parameters: grid, block and shared memory size. They can either be created explicitly by the user and passed to CudaService at the kernel launch or implicitly by the service itself given the total threads the kernel should be executed with (problem size), if the kernel expects a **single-dimensional** configuration. For multidimensional launches, the ExecutionPolicy has to be created explicitly. There is a factory functor, cuda::AutoConfig()(<problem size>), which utilizes a heuristic that attempts to minimize idling hardware resources during the kernel's execution. Example:

```
auto execPol= cuda::AutoConfig()(n, (void*)kernel);
```

ExecutionPolicy and automatic configuration have been taken from Mark Harris' HEMI (open source on Github).

2.1.4 API

The CudaService class provides a simple API to launch GPU tasks:

```
CudaService::cudaLaunch(launchType, kernelWrapper, args...)
```

The parameters are:

- launchType:
  - An explicit cuda::ExecutionPolicy object: for providing a manual launch configuration. This is necessary in case of a multidimensional Grid or Block.
  - unsigned int: The size of the problem (or the total threads the kernel should be run with), in case of automatic launch configuration. To perform an automatic launch configuration, a different flavour of the wrapper should be used. For examples, see the unit tests in the CMSSW fork repository mentioned previously.
- kernelWrapper: A user-provided function with the syntax that was previously demonstrated.
- args...: The arguments that should be passed to the kernel.

Currently the kernels can accept 3 kinds of C++ types as arguments:

- Arithmetic types: int, float, ...
- Pointers to arithmetic types
- Pointers to structures of user-defined structs that comprise only arithmetic types.
  - [FUTURE]: or cudaPointers to arithmetic types (see next section). Currently, structures that include pointers or cudaPointers don't work. (ArrayOfStructs is possible, StructureOfArrays is not)

It should be stressed that the kernel can only receive by value fundamental, arithmetic types. Structs/classes that contain only arithmetic members need to be passed by pointer. STL data structures, like std::vectors or std::maps, are currently not supported by CUDA. A workaround with the use of the Thrust library is a future goal of this system.

2.2 Inside CudaService

CudaService is a ThreadPool that fulfills a GPU management role.

2.2.1 Lifecycle

As it is a CMSSW service, CudaService is meant to only be constructed implicitly by the Framework and never explicitly by the physics code. In fact, it assumes that there are never more than 1 instances and provides only a default constructor, with deleted copy and move semantics. It is meant to be shared by all running threads and is thus thread-safe. The constructor checks the number of GPUs present on the system and registers callbacks to the CMSSW ActivityRegistry to start and end the pool's worker threads when the Framework signals the postBeginJob and postEndJob transitions respectively. If the constructor is called a second time, a runtime exception will occur [release version only].

2.2.2 State

CudaService incarnates a consumer/producer model:

- Incoming tasks are pushed to a tbb::concurrent_bounded_queue
- Waiting threads consume tasks from the queue

The threads are std::threads and are independent from the TBB scheduler. When the queue is empty, they are waiting idly on the blocking "pop" method of tbb::concurrent_bounded_queue.

3 Conclusion

The proposed system has been tested both with unit tests and with an integration test, which also serve as examples, of specific use patterns and a complete use scenario respectively. There are some latency benchmarks. The integration test consists of:

- Changing some physics code to call CudaService
- Writing a python cfg script for cmsRun
- Running cmsRun with 4 threads
- Comparing the GPU results with CPU

and proves that the system functions as a whole, integrated in CMSSW.

Currently, the system provides the basic functionality that has been described and is almost ready to be actually integrated in CMSSW. What has not been tested is how it performs in a multi-GPU environment and in the context of not exclusively managing the GPU (if there is another process that also uses it). Of course, there are many features that are still under development to improve the system's versatility, documented in the source code.

Documentation

The source code is fully documented and there is a tutorial on the use of CudaService on CERN TWiki. Currently it exists at: https://twiki.cern.ch/twiki/bin/view/Main/CudaService