Towards GPUs in CMSSW

Mark Grimes

24/Aug/2015

University of BRISTOL

Overview

- Lindsey Gray gave a good talk on the motivation for running CMS reconstruction on GPUs (https://indico.cern.ch/event/435682/).
- This talk is more on the practicalities of picking and integrating a framework.
- Tried a couple with CMSSW on lxplus but running on CPUs only. I have no performance comparisons yet.
- Quite text heavy sorry.

Potential technologies

- CUDA - nVidia
  - Apparently Alessandro Degano has used it for kd-tree studies.
  - Vendor specific - I've basically ignored it for this reason so don't know much about it. Can't use coprocessors like Intel Xeon Phi (AFAIK).
- OpenCL - Khronos group
  - Open standard from many vendors, can target GPUs, coprocessors, CPUs.
  - Pretty well established.
  - More later...
- Vulkan - Khronos group
  - New standard to replace (?) OpenGL (graphics on GPUs).
  - Tagline "Graphics and compute belong together".
  - Other than the tag line, can find no information about using it for computation in any talks or documentation.
- SYCL - Khronos group
  - New standard that builds on top of OpenCL.
  - C++11 kernels (with some restrictions) in the same source as host code.
  - More later…

Potential technologies

Some others I don’t know much about:

- OpenACC
    - Toolkit available from nVidia (presumably only targets nVidia GPUs).
    - Preliminary support in GCC 5.1 (quite basic).
    - Looks simple (e.g. pragma in source before “for” loop to run on GPU). Not sure about the power (controlling data in and out etcetera).
- C++ AMP - Microsoft
    - Open standard but currently the only implementation is Windows only.
    - Haven’t looked in great detail for this reason.
- OpenHMPP
    - Apparently uses pragmas in a similar way to OpenACC.
    - Implementations (AFAIK) are only in expensive commercial compilers.
- Boost.Compute - boost org (but not official package yet?)
    - Basically OpenCL but more convenient?
- Thrust
    - Only supports nVidia GPUs. I think there is a tbb backend.

OpenCL

- First tried working OpenCL into Pandora (reconstruction toolkit used by HGCal).
- For those that don't know - OpenCL has "host code" that talks to the GPUs and submits "kernels" written in a C subset which run and do the work. Kernel can also run on the CPU.
- Input must be explicitly copied from the host to the GPU and the output copied back again - there is an intrinsic overhead (not specific to OpenCL).
- To cut a long story short - fiddly to setup but works fine for input and output of simple arrays of simple datatypes.
- If we want anything more complicated (even structs) then there is a lot to worry about:
    - Would need a C++ data representation and C kernel side representation.
    - Data packing and padding are not necessarily the same on both devices!!!
    - Would need to explicitly specify struct padding in the kernel code.
    - Sounds like a nightmare to maintain.

SYCL

- Pronounced “sickle”. No idea what it stands for.
- AMD have an experimental open source implementation [triSYCL], but only runs on CPUs (OpenMP underlying).
- Codeplay have a full implementation to be released “later in 2015” [ComputeCpp].
    - Possible to get free previews with a confidentiality agreement.
    - No idea what license options the released version will have.
- No way of knowing what other implementations are yet to be announced.
- Masters thesis [Trigkas] - “SYCL does great from a programmability perspective. However, its performance still does not match that of OpenCL or OpenMP”
    - They used early preview of Codeplay compiler on Xeon Phi. Hard to imagine performance is still so poor.

SYCL

- C++11 based kernels in the same source file as host code.
- Some limitations - no function pointers, recursion, virtual functions or global variables within kernels.

- triSYCL uses C++14, incompatible with GCC used in 6_2_X_SLHC. Simple tests in 7_6_0_pre3 to calculate the Cartesian distance between hits.
    - Would need 6_2_X_SLHC with more recent GCC or Pandora in 7_6_X to do any “real world” performance study (and ComputeCpp instead of triSYCL).

“Hello CMSSW” in SYCL

```
void TestAnalyser::analyze( const edm::Event& event, const edm::EventSetup& eventSetup )
{
	edm::Handle<reco::PFRecHitCollection> hRecHits;
	event.getByToken( inputToken_, hRecHits );

	size_t inputSize=hRecHits->size();
	cl::sycl::queue myQueue;

	cl::sycl::buffer<reco::PFRecHit> inputBuffer( hRecHits->data(), hRecHits->size() );
	cl::sycl::buffer<double> outputBuffer( inputSize*inputSize );

	myQueue.submit( [&]( cl::sycl::handler& myHandler )
	{
		auto inputAccess=inputBuffer.get_access<cl::sycl::access::read>(myHandler);
		auto outputAccess=outputBuffer.get_access<cl::sycl::access::write>(myHandler);

		myHandler.parallel_for<class MyKernel>( inputSize*inputSize, [=](int index)
		{
			const auto& hitPosA=inputAccess[index%inputSize].position();
			const auto& hitPosB=inputAccess[index/inputSize].position();
			outputAccess[index]=std::sqrt( (hitPosA.x()-hitPosB.x())*(hitPosA.x()-hitPosB.x())
			                             + (hitPosA.y()-hitPosB.y())*(hitPosA.y()-hitPosB.y())
			                             + (hitPosA.z()-hitPosB.z())*(hitPosA.z()-hitPosB.z()) );
		});

	});

	auto outputAccess=outputBuffer.get_access<cl::sycl::access::read,cl::sycl::access::host_buffer>();
	size_t indexA=5, indexB=8;
	if( inputSize>8 ) edm::LogInfo("SYCLTest") << "The Cartesian distance between two arbitrary hits is "
			<< outputAccess[indexA*inputSize+indexB];
}
```

“Hello CMSSW” in SYCL

```
void TestAnalyser::analyze( const edm::Event& event, const edm::EventSetup& eventSetup )
{
	edm::Handle<reco::PFRecHitCollection> hRecHits;
	event.getByToken( inputToken_, hRecHits );

	size_t inputSize=hRecHits->size();
	cl::sycl::queue myQueue;

	cl::sycl::buffer<reco::PFRecHit> inputBuffer( hRecHits->data(), hRecHits->size() );
	cl::sycl::buffer<double> outputBuffer( inputSize*inputSize );

	myQueue.submit( [&]( cl::sycl::handler& myHandler )
	{
		auto inputAccess=inputBuffer.get_access<cl::sycl::access::read>(myHandler);
		auto outputAccess=outputBuffer.get_access<cl::sycl::access::write>(myHandler);

		myHandler.parallel_for<class MyKernel>( inputSize*inputSize, [=](int index)
		{
			const auto& hitPosA=inputAccess[index%inputSize].position();
			const auto& hitPosB=inputAccess[index/inputSize].position();
			outputAccess[index]=std::sqrt( (hitPosA.x()-hitPosB.x())*(hitPosA.x()-hitPosB.x())
			                             + (hitPosA.y()-hitPosB.y())*(hitPosA.y()-hitPosB.y())
			                             + (hitPosA.z()-hitPosB.z())*(hitPosA.z()-hitPosB.z()) );
		});

	});

	auto outputAccess=outputBuffer.get_access<cl::sycl::access::read,cl::sycl::access::host_buffer>();
	size_t indexA=5, indexB=8;
	if( inputSize>8 ) edm::LogInfo("SYCLTest") << "The Cartesian distance between two arbitrary hits is "
	        << outputAccess[indexA*inputSize+indexB];
}
```

Access token à la edm::EDGetTokenT. Allows SYCL to automatically copy the required data to the GPU.

This part is the kernel that would run on the GPU

Access tokens also allow the queuing system to calculate dependencies between kernels. E.g. It knows a kernel with read access to `outputBuffer` would have to run after this one.

Data conversion in SYCL

- Probably don’t want full CMSSW data formats on the limited memory space of the GPU (?)
- Conversion through implicit copy construction is easy.
- Apparently all format conversion (presumably including struct padding) done automatically.

```
namespace
{
	struct SimplifiedHit
	{
		float x;
		float y;
		float z;
		float energy;
		SimplifiedHit() = default;
		SimplifiedHit( const reco::PFRecHit& hit ) :
			x(hit.position().x()), y(hit.position().y()), z(hit.position().z()), energy(hit.energy())
		{ /* No operation besides initialiser list */ }
	};

} // end of the unnamed namespace


void TestAnalyser_compactFormat::analyze( const edm::Event& event, const edm::EventSetup& eventSetup )
{
	edm::Handle<reco::PFRecHitCollection> hRecHits;
	event.getByToken( inputToken_, hRecHits );

	...

	cl::sycl::buffer< ::SimplifiedHit > inputBuffer( hRecHits->begin(), hRecHits->end() );

	...
}
```

Data format agnosticism in SYCL

- Could template out accessor functions and provide a library of algorithms.

SomeAlgorithmLibrary.h

```
namespace algorithms
{
    template<class T> inline auto getX( const T& var ) { return var.x; }
    template<class T> inline auto getY( const T& var ) { return var.y; }
    template<class T> inline auto getZ( const T& var ) { return var.z; }

    template<class T_input,class T_output>
    auto makeTestFunction( T_input& inputAccess, T_output& outputAccess, size_t inputSize )
    {
        return [&](int index)
        {
            const auto& hitA=inputAccess[index%inputSize];
            const auto& hitB=inputAccess[index/inputSize];
            outputAccess[index]=std::sqrt( (getX(hitA)-getX(hitB))*(getX(hitA)-getX(hitB))
                                         + (getY(hitA)-getY(hitB))*(getY(hitA)-getY(hitB))
                                         + (getZ(hitA)-getZ(hitB))*(getZ(hitA)-getZ(hitB)) );
        };
} //
```

Provide reasonable default.

TestAnalyser.cc

```
#include "SomeAlgorithmLibrary.h"

namespace algorithms
{
    template<> inline auto getX<reco::PFRecHit>( const reco::PFRecHit& hit ) { return hit.position().x(); }
    template<> inline auto getY<reco::PFRecHit>( const reco::PFRecHit& hit ) { return hit.position().y(); }
    template<> inline auto getZ<reco::PFRecHit>( const reco::PFRecHit& hit ) { return hit.position().z(); }
}

void TestAnalyser_templatedAccess::analyze( const edm::Event& event, const edm::EventSetup& eventSetup )
{
    ...
    myQueue.submit( [&]( cl::sycl::handler& myHandler )
    {
        ...
        myHandler.parallel_for<class MyKernel>( inputSize*inputSize, algorithms::makeTestFunction(inputAccess,outputAccess,inputSize) );
    });
    ...
}
```

Specialise template if required for your datatype.

Framework agnosticism

- Can even write a simple wrapper to use the same algorithm in Thread Building Blocks.

SomeAlgorithmLibrary.h

```
#include <tbb/tbb.h>

namespace algorithms
{
	template<class T>
	auto wrapForTbb( const T& function )
	{
		return [&]( tbb::blocked_range<int> range )
		{
			for( int index=range.begin(); index!=range.end(); ++index )
			{
				function(index);
			}
		};
	}
} // end of namespace "algorithms"
```

TestAnalyser.cc

```
#include "SomeAlgorithmLibrary.h"

void TestAnalyser_templatedAccess::analyze( const edm::Event& event, const edm::EventSetup& eventSetup )
{
	edm::Handle<reco::PFRecHitCollection> hRecHits;
	event.getByToken( inputToken_, hRecHits );

	size_t inputSize=hRecHits->size();
	std::vector<double> tbbOutput( inputSize*inputSize );

	tbb::parallel_for( tbb::blocked_range<int>(0,inputSize*inputSize),
		algorithms::wrapForTbb( algorithms::makeTestFunction(*hRecHits,tbbOutput,inputSize) ) );

	...
}
```

- Although note this example wastes CPU because it calculates A→B, but also B→A and A→A etcetera (N^2 instead of N(N-1)/2).
- Some algorithms fundamentally different on GPU to CPU.

Summary and next steps

- SYCL shows a lot of promise but still needs to be fully assessed. Assessing on an actual GPU would be nice.
    - Trying to sort out bureaucracy to evaluate Codeplay compiler.
    - Not sure how easy it is going to be to work into scram.
- I don't know enough about the other frameworks to completely rule them out.
- Whatever happens, it looks like the field is moving towards C++11 based kernels (SYCL, OpenCL 2.1, C++ AMP, ...).
    - Templated algorithm libraries look the easiest way to future proof ourselves.
    - But try to keep in mind limits (no recursion etcetera).
- What GPU ready resources do we have?
    - Bristol has a farm with some GPU queues, can use that for initial testing. Probably need Cern resources for much else.
- Try to find others with experience. Found out last week Bristol Uni is an Academic Member of Khronos (Computer Science department).

- [Trigkas] Angelos Trigkas, University of Edinburgh, August 2014, https://static.ph.ed.ac.uk/dissertations/hpc-msc/2013-2014/Investigation%20of%20the%20OpenCL%20SYCL%20Programming%20Model.pdf
- [triSYCL] https://github.com/amd/triSYCL
- [ComputeCpp] https://www.codeplay.com/products/computecpp