SYCL For HEP: Experimenting with GPU Kernels in Pythia8. (Draft)

Introduction to SYCL

The SYCL [https://www.khronos.org/registry/SYCL/specs/sycl-1.2.1.pdf] specification by the Kronos group is an abstraction layer, allowing for a single code base to compiled and executed on multiple different heterogeneous computing architectures. As opposed to writing specific OpenCL, CUDA, etc. implementations.

SYCL code is written in C++ and allows the use of language features such as templation and lambda function, while hiding in the abstract layer many of the underlying complexities of deploying an OpenCL style application. It allows for the easy integration of accelerated code within larger, existing, C++ frameworks.

ComputeCpp is a full implementation of the SYCL 1.2.1 specification produced by CodePlay [https://developer.codeplay.com/products/computecpp/ce/home/]. The community edition is free to download and use. Downloading the software requires a free codeplay account registration, the licence permits the redistribution of the libComputeCpp.so runtime which must be linked against by the accelerated application.

At compilation time ComputeCpp isolates and extracts SYCL code found inside the application (i.e. computational kernels), and compiles this to an intermediate format. By default a Standard Portable Intermediate Representation (SPIR-V) is produced (SPIR-V was introduced in 2015 and works as a replacement to SPIR). SPIR and SPIR-V are intermediate representations of OpenCL and as such are deployable to hardware produced by AMD, Intel and ARM.

While Nvidia GPUs do provide an OpenCL interface, they do not provide a SPIR-V interface. Hence to target Nvidia hardware, ComputeCpp compiles instead to the Parallel Thread Execution (PTX) binary format. PTX is the intermediate representation of the CUDA language.

A limitation of the free community edition is the ability to only target a single representation (SPIR, SPIR-V or PTX), Multi-Binary Support is a feature of the professional edition.

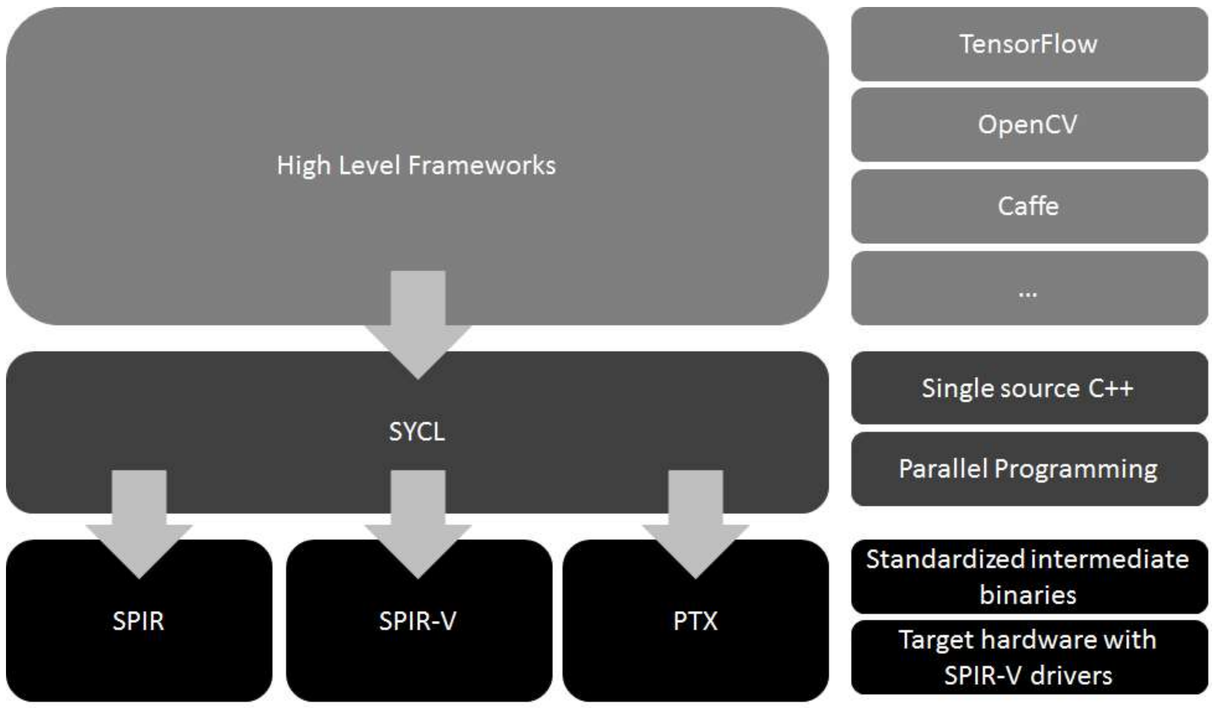

Fig 1 [Image credit: https://www.codeplay.com/portal/news/2017/12/06/adding-experimental-ptx-support-to-computecpp-for-nvidia-reg-hardware.html]

Relationship between SYCL and OneAPI

OneAPI [https://oneapi.com] builds on top of ISO standard C++17 and Khronos SYCL to additionally add Data Parallel C++ (DPC++) to the specification. This adds functionality such as Unified Shared Memory. Both Intel and Codeplay develop OneAPI toolchains, and in addition there is an open source project. The OneAPI specification covers multiple APIs which specialise in video processing, deep neural networks, etc.

Potential benefits and limitations

In the following we will make the assumption that SYCL based acceleration should be a purely optional component of a HEP software package or library which may be enabled with flags at compile time, and subsequently executed on a system with compatible OpenCL drivers.

C++ is widely used in performant HEP software, SYCL code can be used to augment existing packages by offloading specific computations to heterogeneous processing devices.

This can theoretically allow a single block of C++ code to be executed natively by the CPU (i.e. not using SYCL), or to be compiled as part of a SYCL kernel running in either Host Mode or Device Mode. Here Host mode refers to an emulated backend which allows for the execution of device code in native C++. Device mode dispatches at runtime to an available accelerator, which will be one of GPU, Accelerator or CPU - depending on hardware and driver availability.

The Host mode is not designed for production systems, the OpenCL device emulation adds significant overheads. But it does allow SYCL kernels to be debugged using standard tools such as gdb.

There are two main considerations when attempting to reuse code inside both a standard C++ function call and a SYCL kernel.

- Memory access model. The input and output data from the computation must be shipped to and from the execution device. The input data should therefore ideally be stored in memory as a contiguous array of an appropriate data structure. Either a C-style array, or a single dimensional `std::vector`. An array of the `vec<double, 4>` structure is used in the example below to represent the 4-vectors of a set of particles.
- Computation model. The optimal design of an algorithm designed to process a large task sequentially may differ somewhat from that of the same algorithm designed to process a large task concurrently. For example, the sequential design may use branching logic to avoid performing an expensive computation when it can deduce that the result would not be useful. Whereas deployed on a GPU, all execution units within a given group must execute in lockstep with each other. Here long conditional branches, or variable length loops are to be avoided. With the greatest throughput obtained for a fixed-length computation over all permissible inputs.

As a result of these constraints, the easiest functionality of the application to convert such that it operates efficiently when used within SYCL kernels is that which corresponds to the application's lower-level classes, and utility classes. Examples would include a 4-vector class, and a transformation matrix class to operate on 4-vectors. As a side note, SYCL interface to the Eigen matrix library is available.

When it then comes to integrating the main computational function within the SYCL kernel, the choice must then be made between.

- Providing two implementations of the function, one optimised for sequential / CPU processing and another optimised for parallel processing within a kernel.
- Provide one implementation but use preprocessor macros to make (small) modifications to the source code depending on the compiler.
- Provide one implementation which is efficiently usable under both execution paradigms.

This third category is likely rare, so most functions will likely use either of the first two categories - which have respective trade offs in terms of code comprehension and fragmentation.

Integrating SYCL with existing HEP software

In this example the Pythia8 Monte Carlo version 8.303 [http://home.thep.lu.se/~torbjorn/Pythia.html] will be built from source on CentOS7 using the gcc 7.5 toolchain. The build process will be integrated with CodePlay's ComputeCpp v.1.2.1.

Pythia8 uses a GNU style build system (i.e. ./configure, followed by make), with optional external libraries supplied during the configure phase. Documentation is provided by CodePlay on the integration of both GNU build and CMake build configurations, covering the majority of HEP software.

SYCL support requires that the following externals are available:

- OpenCL API Headers from the Khronos Group [https://github.com/KhronosGroup/OpenCL-Headers]
- ComputeCpp Community Edition SDK [https://developer.codeplay.com/products/computecpp/ce/download], here using ComputeCpp-CE-2.1.0-x86_64-linux-gnu. This provides:
    - SYCL include headers.
    - `libComputeCpp.so` (the SYCL runtime).
    - `compute++` binary (the SYCL device compiler), and other helper binaries such as `computecpp_info`
- gcc 4.8 or higher
- CMake 3.4.3 or higher (for cmake applications, not used in the following example).

While the build process can be modified to only use `compute++`, SYCL support is more commonly enabled via a multi-compiler approach as illustrated below, using both `compute++` and the gcc supplied `g++` compiler.

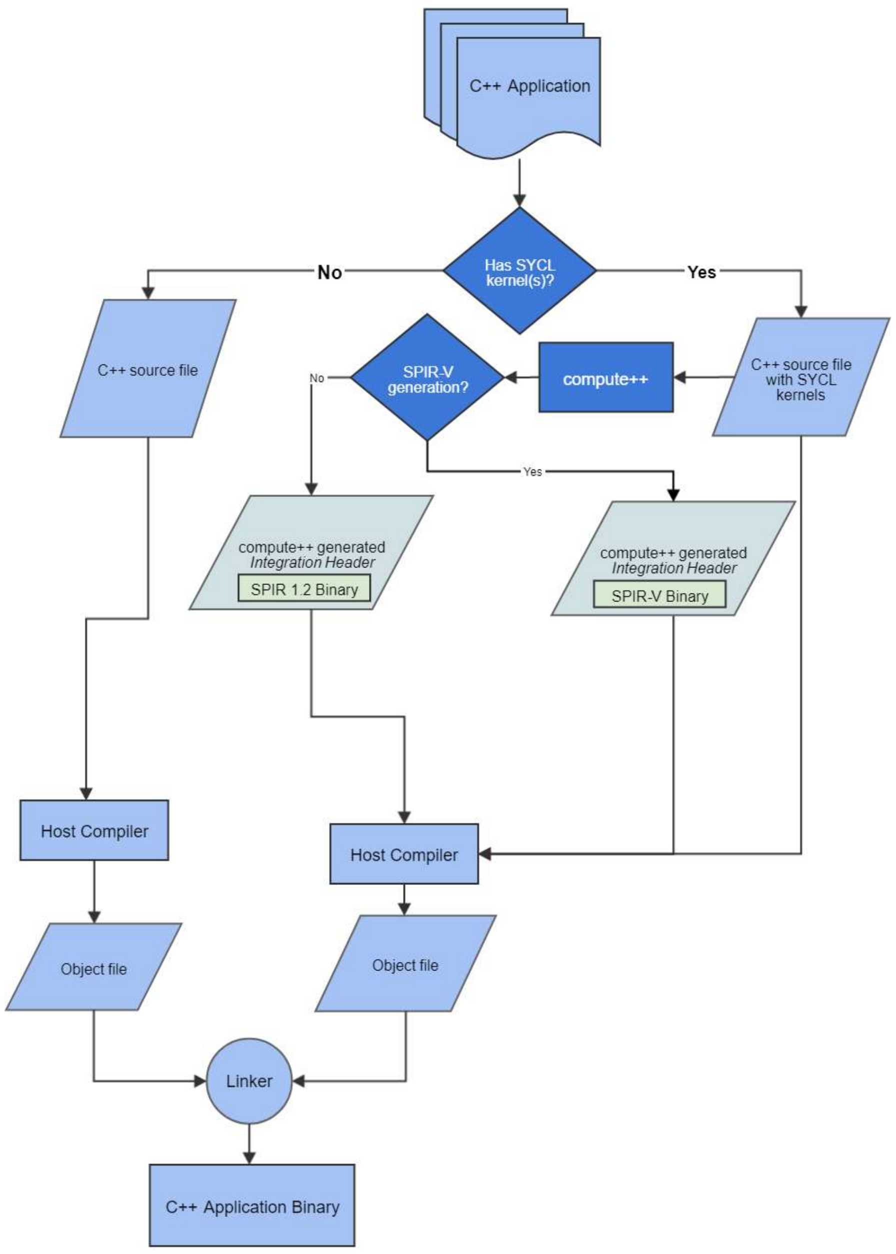

Fig 2 [Image credit https://developer.codeplay.com/products/computecpp/ce/guides/integration-guide]

The device compiler, `compute++`, is used to generate binaries in SPIR, SPIR-V or PTX (not shown in figure) format for any kernels found within each source file. This procedure outputs an integration header which is included in Pythia8's standard object compilation. For source files without any kernels, the integration header is empty. This secondary host compilation allows ComputeCpp to access the compiled kernels at runtime.

In practice, this requires the generation and use of an intermediate compilation fragment, the integration header with suffix .sycl

Make fragment. Generate `src/file.sycl` from `src/file.cc`:

```
$(LOCAL_SRC)/%.sycl: $(LOCAL_SRC)/%.cc
```

```
     $(COMPUTECPP) ${COMPUTECPP_DEVICE_COMPILER_FLAGS} $< -o $@ -c
$(OBJ_COMMON) -I$(ComputeCpp_INCLUDE_DIR) -I$(OpenCL_INCLUDE_DIR)
```

Here **`$(COMPUTECPP)`** is the path to the `compute++` binary, **`$(OBJ_COMMON)`** are the common `g++` compiler flags, **`$(ComputeCpp_INCLUDE_DIR)`** and **`$(OpenCL_INCLUDE_DIR)`** are the respective CompureCpp and OpenCL header directories, and **`$(COMPUTECPP_DEVICE_COMPILER_FLAGS)`** is set to
`-sycl-target ptx64 -sycl-device-only`
`--gcc-toolchain=/home/epp/phsmai/sycl/gcc/gcc-7.5.0_install`
Here we note that the ptx64 intermediate representation is specified - targeting Nvidia hardware, and the gcc installation is specified explicitly as the system installation is not used in this example.

The integration header is then injected into Pythia8's normal object compilation.

Make fragment. Generate `tmp/file.o` from both `src/file.cc` and `src/file.sycl`. Force `file.sycl` to be included at the top of file.cc. The SYCL additions are highlighted in red.

```
$(LOCAL_TMP)/%.o: $(LOCAL_SRC)/%.sycl $(LOCAL_SRC)/%.cc
     $(CXX) -include $^ -o $@ -c $(OBJ_COMMON)
```

Finally, the linker command is modified such that `libpythia8.so` is linked to `libComputeCpp.so`

Linker modification. The SYCL additions are highlighted in red.

```
LIB_COMMON=-Wl,-rpath,$(PREFIX_LIB) -Wl,-rpath,$(ComputeCpp_LIB_DIR)
-L$(ComputeCpp_LIB_DIR) -lComputeCpp -ldl $(GZIP_LIB)
```

With these modifications in place, Pythia authors may start to add kernels into the event generator. The SYCL runtime will activate for any main executable built against this pythia8 shared library.

Kernel dispatch is handled at runtime based on the OpenCL drivers available on the execution system. Given multiple choices, the kernel programmer may decide which device is chosen for dispatch using various heuristics (e.g. favour GPUs). The parallelisation capacity of the device can also be queried and used to optimise the chunking of the computation.

The computecpp_info helper binary prints information about compatible devices on a given system, the output of the application on the development machine is:

```
Device 0:
  Device is supported       : UNTESTED - Vendor not tested on this OS
  Bitcode targets           : ptx64
  CL_DEVICE_NAME            : Quadro P1000
  CL_DEVICE_VENDOR          : NVIDIA Corporation
```

```
CL_DRIVER_VERSION          : 450.80.02
CL_DEVICE_TYPE             : CL_DEVICE_TYPE_GPU
```

For systems with zero hardware devices, the Host mode may be used as a fallback. As noted above, this provides a software emulated OpenCL execution environment which is designed for testing and debugging, it is not designed to be used in production.

If no supported hardware devices are available in a production job, it is always preferable to perform the computation natively (i.e. as if SYCL was not compiled into Pythia8) than it would be to dispatch in Host mode.

The following table presents matrix multiplication times for different matrix sizes, as computed using a native C++ implementation running on a Intel(R) Core(TM) i7-7700 CPU @ 3.60GHz with no explicit parallelisation, or a SYCL kernel dispatched to either the Quadro P1000 GPU or to the Host mode emulator.

GPU acceleration on this entry-level card is noted to be around 100 times faster than CPU computation for large matrices. For small matrices, the performance gain is lost due to the overhead of GPU dispatch. Host mode, on the hand, is observed to be around 100 times slower than CPU computation, this is due to the overhead of emulating an OpenCL environment.

Matrix Size [NxN]	**Host Mode Time [ms]**	**Host Mode GFlops**	**CPU Time [ms]**	**CPU GFlops**	**P1000 GPU Time [ms]**	**P1000 GPU GFlops**
64	180	$3\text{x}10^{-3}$	1	0.52	6	0.09
256	6,900	$5\text{x}10^{-3}$	110	0.32	26	1.29
1024	370,000	$6\text{x}10^{-3}$	6,900	0.31	66	32.5
2048	Not Run	Not Run	54,700	0.31	500	34.4

Table 1: CPU time in milliseconds and and giga floating point operations per second to multiply two symmetric matrices.

Practical example of SYCL integration

Pythia 8 contains a full suite of models allowing it to simulate the multiple soft and semi-hard scatters (MPI) which occur within a proton-proton interaction in addition to the hard scatter. These, in addition to the beam remnant, undergo a colour reconnection simulation before the final state is frozen out in hadronisation.

Simulating 1,000 QCD events at sqrt(s) = 14 TeV (p_T min 20 GeV) in Pythia 8.303 using a single 3.6 GHz CPU core and the default tune takes 5.1 seconds. This time is relatively insensitive to the scale of the hard interaction, increasing p_T min to 2 TeV increases the overall time by 12%, to 5.7 seconds.

More sophisticated models such as [https://link.springer.com/article/10.1007/JHEP08(2015)003] allow for colour reconnections to be computed beyond leading colour, including a realistic treatment of the SU(3) colour structure. This alternate simulation provides significantly better physics modelling of the soft MPI-generated structure of the event when compared to LHC data. It is enabled by setting both `ColourReconnection` and `BeamRemnants` parameters to mode 1 [http://home.thep.lu.se/~torbjorn/pythia82html/ColourReconnection.html#section1].

The equivalent timings for 1,000 events using the beyond leading colour model are 22.3 seconds for p_T min of 20 GeV, and 27.0 seconds for pT min of 2 TeV. These correspond to a factor x4.4 and x4.7 (respective) increase in CPU cost to simulate a given number of events.

Flame graphs are presented below for the default and beyond leading colour models for a p_T min of 2 TeV. The X-axis corresponds to the total CPU time of the job, the call stack is visualised on the Y-axis.

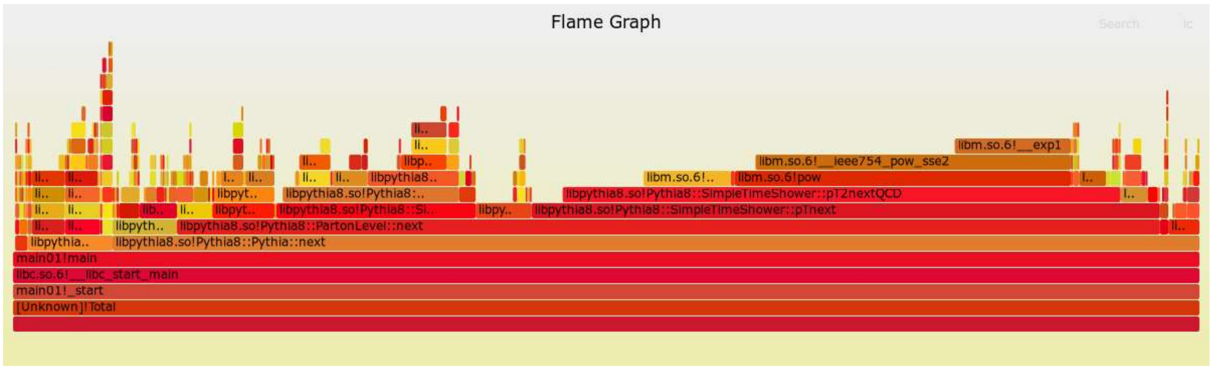

Fig 3: Flamegraph of default MPI modelling, covering 5.7 second. Only 0.4% of CPU time is spent under **`Pythia8::ColourReconnection::next`**. The heaviest function under **`Pythia8::PartonLevel::next`** is **`Pythia8::SimpleTimeShower::pTnext`**, at 53% (2.7 ms/event).

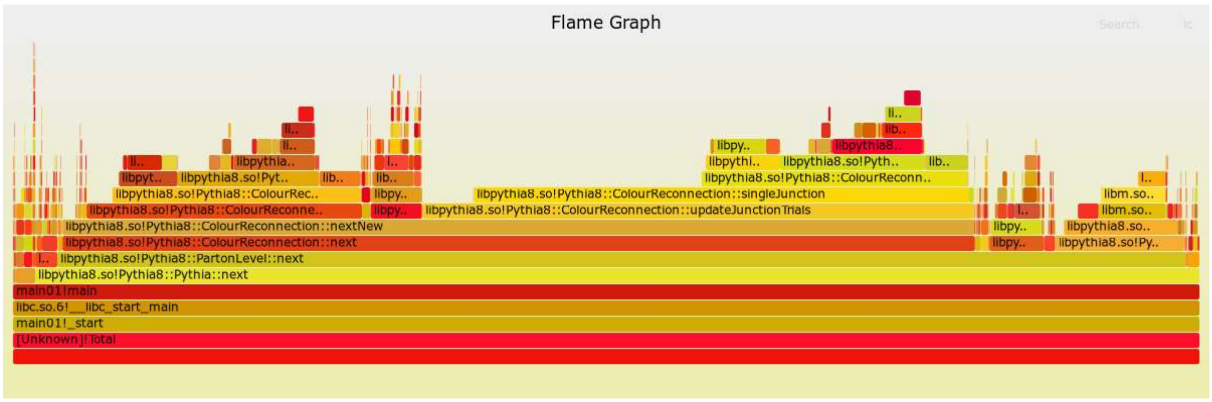

Fig 4: Flamegraph of beyond leading colour modelling, covering 27 seconds. 77% of CPU time is spent under **`Pythia8::ColourReconnection::next`** (20 ms/event). 11% of CPU time is spent under **`Pythia8::SimpleTimeShower::pTnext`** (3 ms/event), comparable to the default case.

Inspection of the expensive **`Pythia8::ColourReconnection::next`** call reveals the most expensive call to be **`Pythia8::ColourReconnection::singleJunction`**. This is called both directly, and within **`Pythia8::ColourReconnection::updateJunctionTrials`**. These calls consume 23% and 42% of CPU, respectively, or 18 ms/event combined.

The computation sets up vectors of pseudoparticle and dipole objects, dipole objects reference pseudoparticle objects by index. The **`singleJunction`** function performs trial reconnections on either two or three input dipoles using the following logic.

The dipoles are subdivided into three roughly equal sized subsets by taking the modulo 3 on each dipole's colour index. The two-dipole **`singleJunction`** function is called over each combination of indices, excluding identical and interchanged indices.

Example: for 4, 6 and 5 dipoles in each subgroup, a total of 31 calls are made. Corresponding to the upper off-diagonal elements in blue.

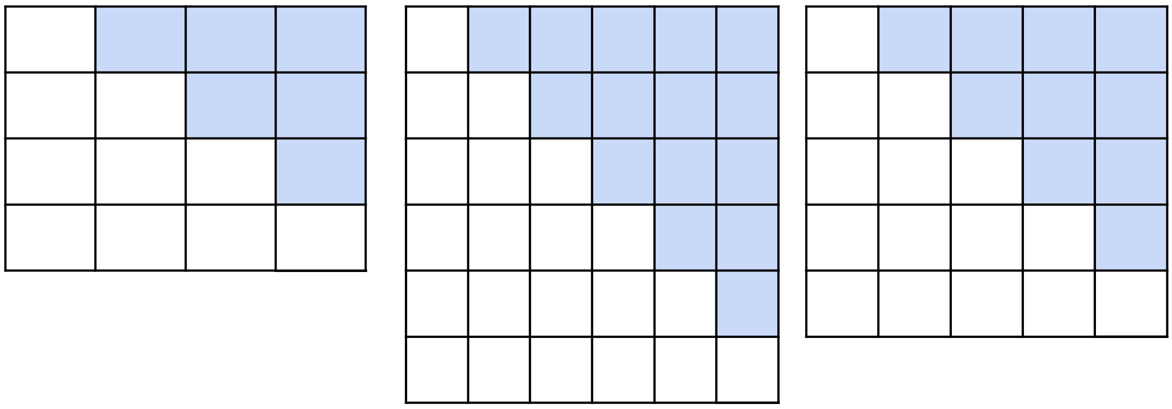

Fig 5: For real events, a mean of 4,900 calls are made to the two-dipole function per iteration, with a standard deviation of 3,400. The minimum and maximum calls were 212 and 23,080.

The three-dipole **`singleJunction`** is called over all triplets (i, j, k) of indices within each subset, where i runs from 0 to N, j runs from (i+1) to N, and k runs from (j+1) to N. Here N is the number of dipoles in the subset. For real events, a mean of 107,000 calls are made to the three-dipole function per iteration, with a standard deviation of 115,000. The minimum and maximum calls were 749 and 962,520.

The key quantity computed over each call to **`singleJunction`** is LambdaDiff, a floating point parameter corresponding to the difference in the string-length and junction-length between the dipoles. Each combination which is found to be energetically favourable to form a junction (LambdaDiff is positive) is added to a list of trial reconnections for further processing.

Parallelisation of singleJunction with SYCL

By encapsulating the logic of **`singleJunction`** in a SYCL kernel, the above parameter space may be explored in parallel.
For this exercise, the input data will be copied into contiguous memory regions. However the existing class structure used within Pythia8 is already very close to being directly usable without format shifting. The time taken to perform the format-shift will be accounted for separately.

In the following, **`sycl`** is a typedef of **`cl::sycl`**, and **`sycl::double4`** is a typedef of **`sycl::vec<sycl::cl_double, 4>`**. Here 'vec' refers to a geometric vector. SYCL versions of primitives, such as **`sycl::cl_double`**, are used to maintain consistent definition between the host and device.

Only four-vector data from the list of pseudoparticles in the event are required in the computation of lambdaDiff, the data are copied into a C-style array **`sycl::double4 particles[N]`** whose memory is assigned on the heap. This allocates storage for four double precision inputs (i.e. a **`{0., 0., 0., 0.}`** struct) for each of the N particles, the particles four-momentum are copied into this array. Alternatively, Pythia8's **`Vec4`** class could be used directly inside SYCL kernels. This would require minimal modifications to Pythia.

Similarly memory is allocated for each dipole, here a **`sycl::int4`** is used to store indices to two particles, a colour reconnection index, and a status index. A **`sycl::double`** is used to store the dipole's creation time and a **`sycl::double4`** is used to store the dipole's four momentum.

The final input corresponds to the slicing of the dipole data into three subsets. This is stored in a **`std::vector<std::vector<sycl::cl_int>>`** format, where the outer vector is of size three and the inner vectors are of variable size, these contain indices within the dipole memory structure. The inner vector's memory is used directly within SYCL, with no additional manipulations required (beyond the use of the SYCL defined primitive type).

Finally the results of the three two-dipole computations and the three three-dipole computations are stored within six allocated regions of memory. E.g. for the results of the three two-dipole computations:

```
std::unique_ptr<sycl::cl_double[]> lambdaDiff_2dipole_sycl[3];
for (size_t i = 0; i < 3; ++i) {
  const size_t dipsInSubset = dipSubsets.at(i).size();
  lambdaDiff_2dipole_sycl[i] =
    std::make_unique<sycl::cl_double[]>(dipsInSubset
    * dipsInSubset);
}
```

Kernel Setup

First a device queue is constructed. This encapsulates the OpenCL context, and handles the full lifecycle of kernel submission and execution.

```
sycl::queue deviceQueue(sycl_selector, exception_handler);
```

The optional **`sycl_selector`** is an implementation of **`sycl::device_selector`**. This is supplied with a list of the devices available on the machine - made accessible via platforms (drivers). The selector choses which device will be used for dispatch. The **`exception_handler`** allows for the reporting of asymmetric exceptions, which may be thrown from a different thread than the one executing Pythia.

SYCL requires exclusive access to the relevant ranges of host memory during kernel execution. Some data within these ranges may need to be shipped to external processing devices, and other data may need to be overwritten with returned values.

This is obtained via the use of buffers which bind to supplied addresses. Each buffer takes ownership during its lifespan under the RIIA paradigm. The buffers are hence created within a new scope to control their lifespan.

```
{
  sycl::buffer<sycl::int4, 1> buf_dipoles(dipoles_sycl.get(),
    sycl::range<1>(dipoles_sycl.size()));
  sycl::buffer<sycl::double4, 1>
    buf_particles(particles_sycl.get(),
    sycl::range<1>(particles.size()));
  // ... Other buffers
```

The 1 in both the buffer and range template parameters indicate that the data represent a one dimensional array.

Similar 1D input buffers are setup over each of the three subsets of dipole indices, and 2D output buffers over the six output memory allocations.

The execution of a kernel is encapsulated using a command group handle (**`cgh`**). The encapsulation occurs inside a lambda function which captures the buffers by reference.

First **`sycl::accessor`** objects are constructed which declare the data dependency of the kernel on the buffers. A **`read`** dependency copies the data to the external device, a **`discard_write`** dependency only copies the data from the device back to the host - overwriting whatever happened to be there before. Only global device memory is used in this example.

Second the **`sycl::range`** of the computation is constructed. For the two-dipole computation, we are computing in a two dimensional space. Up to 3 dimensions are supported.

Finally a **`parallel_for`** kernel is submitted over this range. This takes the form of another lambda function, this captures by value.

```
for (size_t i = 0; i < 3; ++i) {
  deviceQueue.submit([&] (sycl::handler& cgh) {
    auto acc_dipoles =
      buf_dipoles.get_access<sycl::access::mode::read>(cgh);
    // ... Other mode::read inputs
    auto acc_lambdaDiff =
      buf_lambdaDiff_2dipole[i].get_access<
      sycl::access::mode::discard_write>(cgh);

    cl::sycl::range<2> range(dipSubsets.at(i).size(),
      dipSubsets.at(i).size());

    cgh.parallel_for<SJKernel>(range, [=](sycl::id<2> item) {
      sycl::cl_int dip1 = acc_dips[item[0]];
      sycl::cl_int dip2 = acc_dips[item[1]];
      acc_lambdaDiff[item] = getLambdaDiffMode0(
        dip1, dip2, &acc_dipoles[0], &acc_creationtimes[0],
        &acc_jmom[0], &acc_particles[0]);
    });
  });
}
```

The equivalent of the above is done also for the three three-junction kernels.

The class **`SJKernel`** is a dummy class name, it is only used to associate the C++ lambda function to the compiled device kernel. The main logic of the execution is held here within the **`getLambdaDiffMode0`** function. This function is provided with pointers to both the dipole data array and the particle data array (when executing on the remote device, the & operator applied to the accessor correctly maps to the remote devices's memory, here global memory on the GPU), along with the two dipole indices for which the lambda difference is to be computed.

All C++ code within this function is executed in parallel on the GPU over the supplied range. The return values, one double per execution, are written to the respective **`buf_lambdaDiff`** buffers.

Separate implementations are used in this example when computing **`singleJunction`** natively vs. on the GPU. This is to explore the effect of different design choices which may be made in the GPU implementation. However one could envisage a setup where the `getLambdaDiffMode0` is instead called within a regular C++ loop in versions of the software which were either compiled without SYCL support, or which do not have access to a suitable OpenCL device at runtime.

The kernel dispatch performed by `deviceQueue` is asynchronous, i.e the call to `submit` returns instantly. SYCL is able to use its knowledge of the data dependency relationship between kernels to correctly order their execution. At present none of the six submitted kernels have any dependency on each other, but a possible extension to this example would be to introduce additional kernels which consume the `buf_lambdaDiff` outputs and perform additional computations on these. This can be used to setup highly efficient chains of kernels with no host-device I/O overheads, as output data produced on-device from one kernel will be read directly by the subsequent kernel.

The main thread running the Pythia8 executable will halt on the closing of the scope containing the buffers. Under the RIIA design, the destruction of each buffer cannot complete until there are no remaining accessors with an open handle on the buffer.

The results of the computation may then be inspected by iterating over the output data array

```
for (size_t subset = 0; subset < 3; ++subset) {
  const size_t dipsInSubset = dipSubsets.at(subset).size();
  for (int i = 0; i < dipsInSubset; ++i) {
    for (int j = i + 1; j < dipsInSubset; ++j) {
      const size_t index = (i * dipsInSubset) + j;
      sycl::double result =
        lambdaDiff_2dipole_sycl[subset].get()[ index ];
    }
  }
}
```

Performance metrics for CPU vs. GPU computation

The mean time required to perform different parts of the computation, not all contribute to the final time as some are subsets or alternate formulations. The per event variance in the time is observed to be greater than that of CPU processing.

For the CPU times in Table 2, these are narrow measures over just the lambda diff and dipole computation functions. In addition, given a dipole pair, the C++ code additionally tries to “walk” out along the string from the given point, performing potentially many more junction attempts within the so-called “single” junction function. This was not implemented in the SYCL version due to the presence of the while loop controlling this. This is why the CPU times listed here are significantly smaller than the 27,000 mu s per event overall CPU time to process each event, listed above.

Native C++ Operation	**Time / Event [mu s]**
Dipole Check (3x 2D)	52
LambdaDiff Compute (3x 2D)	20
Total for 3x 2D	**72**
Dipole Check (3x 2D)	6
LambdaDiff Compute (3x 3D)	257
Total for 3x 3D	**263**

Table 2

SYCL Operation	**Cumulative Time / Event [mu s]**
Host data preparation†	155
& spawn and run three empty kernels	329
& copy particle and dipole data to GPU†	7,620
& copy dipole subsets to GPU†	12,670
& execute Kernels over the Range (3x 2D):	
Empty Kernels	12,567
Compute Kernels ...	166,987
… with early exit after j <= i check	167,108
… with early exit after dipole checks†	165,322
… with early exit after time dilation checks	150,903
Copy results from GPU†	148,073

Table 3. Critical path show with†. Cases where smaller numbers are shown later on are indicative of run-to-run fluctuations, which are larger for GPU dispatch than CPU computation.

The host data preparation covers the copying of the particle 4-vector and dipole data into contiguous host memory. These costs are relatively small here, at around 0.2 ms they are within the run-to-run variation. By further modifying the memory structure used by the native C++ implementation to be compatible with the SYCL implementation, this cost could be reduced further.

The baseline cost to spawn three (empty) 1D kernels over a range of 1 is around 0.3 ms.

A major bottleneck is the time required to send the data required to perform the compute to the GPU. 8 ms are required for the particle 4-vectors (mean size 5,600 bytes) and dipole data (mean size 19,000 bytes). An additional 5 ms are required to send the mapping of the (i, j) indices to be used inside the three kernels to the respective entries inside the dipole data (mean size 600 bytes).

The execution of three *empty* kernels over their full 2D range takes around 13 ms. The empty kernel is then replaced with different variations of the compute kernel. Each variation makes increasing use of branching logic inside the kernel.

Starting at “Compute Kernel”, Lambda Diff is computed for each element of each 2D kernel.

For “Early exit after j <= i check” the kernel only processes (i,j) coordinates above the diagonal, the majority of the work groups will contain work items over (i,j) pairs such that every work item will pass, or every work item will fail. Hence this should remain efficient.

The “Early exit after dipole checks” and subsequent inclusion of “Early exit after time dilation checks”, will both cause a greater amount of fragmentation in the execution flow of work items within the work groups. In theory this should result in a poorer performance as the branching logic inhibits the ability of multiple work items to process in parallel.

We see however that the difference in time between these strategies is similar and within run-to run fluctuation, with the exception of the final exit after time dilation check which is 10% faster.

The final stage is copying back the results in their 2D arrays; this is the largest copy at a mean of 83,600 bytes. The time required is not significant on top of the kernel execution and is hence hard to determine accurately here.

Overall we see that on the entry level Quadro P1000 device, the computation is significantly slower on the GPU. And, even for instantaneous computation, the data transfer overhead alone would make the GPU offloading unproductive at the level described here.

Hybrid Processing

The computational paradigm may be shifted to better use the strengths of each platform.

The three 2D and three 3D loops over the junction space are performed on the CPU, but only to apply either the cheap early rejection, or this plus the more expensive time dilation rejection.

The GPU is still sent all particle and dipole data, but now it is sent three lists of pairs, (i,j), and three lists of triplets (i,j,k), corresponding to the indices which survive the CPU early rejection.

The kernel dimensionality is changed, instead of executing over 2D and 3D ranges, the kernels execute over 1D ranges corresponding to the size of their input list.

The output buffers are also 1D instead of 2D or 3D, they are the same length as their input list. This is where the primary savings in data transfer comes in.

This method becomes more optimal when only sparse entries in the 2D and 3D permutation spaces correspond to valid solutions, and the cost to determine validity is low.

The simple index checks remove 77.4% of the parameter space, adding the expensive time dilation check increases this to 99.8% of the parameter space. Note that the parameter space here is taken as only above the upper diagonal, as on Fig 5. The equivalent numbers for the 3D parameter space are 95.9% and 99.9994%.

Hence the problem could be refactored as simple checks (CPU), time dilation checks (CPU or GPU Kernel), lambda diff (GPU Kernel, with possible dependency on lambda diff kernel).

Conclusions

SYCL allows for single-source C++ to be compiled both for CPU execution and into an intermediate representation of OpenCL (SPIR-V) or CUDA (PTX). Machines with OpenCL drivers may then choose to dispatch the computation to an accelerated device whereas other machines are still able to execute the computation on the CPU.

To accommodate this, the compilation process is modified to first compile the kernel representations with a SYCL compiler (CodePlay’s ComputeCpp community edition) and generate an integration header, the integration header is included by the regular g++/clang/etc. Compilation which allows the runtime to locate the compiled kernel binaries. The final library is linked against the ComputeCpp runtime shared library which permits distribution. Support for Makefile and CMake integrations are documented.

The Kernel dispatch is handled at runtime and can be customised / predicated upon the choice of acceleration devices and their properties. A high level RIAA and data dependency model allows for chains of kernels where the output of one kernel may be the input to another to process in the correct order, maintaining data locality.

Issues common to GPU offloading persist, notably the additional latency for data transfer which competes with any speed up due to parallelisation. Around 10 ms should be budgeted for modest data transfers (O(25 kb)).

While the same C++ can be used for CPU and accelerator deployment, the code must realistically conform to a coding paradigm which is appropriate for accelerator deployment. This may involve some large refactoring projects. The main concerns are I/O, typically via a defined block of memory, and large-scale branching logic - which is to be avoided.

Lower level helper classes (e.g. four vector) generally require a small level of refactoring to be made SYCL compliant, and then may be used inside the dual-use C++ compute functions.