Draft version 1.0

ATLAS NOTE

October 7, 2014

Instructions for the MadGraph OpenCL/CUDA Extension

A. DeAbreu[a], D. Schouten[b], B. Stelzer[a]

[a]*Department of Physics, Simon Fraser University, 8888 University Dr, Burnaby, BC*
[b]*TRIUMF, 4004 Wesbrook Mall, Vancouver, BC*

Abstract

An instructional document for the use of the MadGraph OpenCL/CUDA extension. This extension allows the output of matrix element code for an arbitrary 2→N process in an OpenCL, and CUDA format for compilation and evaluation on Graphics Processing Units or other valid devices. This document will explain the output generated by the extension as well as give a step-by-step walkthrough on the use of these outputs to calculate the matrix element of a process generated by MadGraph on a linux machine. This walkthrough uses Python and the convenient packages, PyOpenCL and PyCUDA, to setup and launch the OpenCL and CUDA kernels.

1 Introduction

General purpose programming on Graphics Processing Units (GPU's) has made powerful parallel computing capabilities readily available. The main languages for GPU programming are OpenCL [1] and CUDA [2] (restricted to GPU's manufactured by NVidia). Both languages are based on C/C++. In both languages a so called kernel is passed from the host side CPU to the GPU or valid device where it is compiled and executed. The result of is then returned from device to the host. The Python packages PyOpenCL and PyCUDA [3] provide convenient interfaces for the loading and unloading of kernels and data to devices.

A extension to the event generator MadGraph 5 (v. 1.5.14) [4] was created in order to output matrix element code in OpenCL, CUDA and standard C++ format for a general $2 \rightarrow N$ process. GPU parallel computing has been used previously for the evaluation of matrix elements in a multidimensional phase space [5] and for a benchmark of Matrix Element Method (MEM) analysis [6]. The work done in [6] uses the MadGraph OpenCL/CUDA extension described in this paper. The MEM was first studied in [7]. A good introduction to the MEM in the context of top mass determination is [8, 9].

The matrix element is calculated using the HELAS functions [10, 11]. The parton distribution functions (PDF's) are evaluated in the kernel using wrapper code that interfaces with LHAPDF [12] and the CTEQ [13] PDF library. The PDF data is stored in (x,Q^2) grids for each parton flavour, where x is the fraction of the beam energy and Q^2 is the momentum transfer. The evaluation of the PDF for an arbitrary point is done using bilinear interpolation within the kernel. The precision of the interpolation is within 1% of the values from directly querying the PDF library. These PDF grids along with the model parameters and event kinematics are transferred to the memory of the device at which point the kernel is executed.

This note is organized as follows: Section 2 contains information on the programs needed to compile and run the output code using OpenCL or CUDA. Section 3 contains instructions for creating the OpenCL/CUDA output from MadGraph 5 as well as descriptions of the generated files. Section 4 contains an in depth description of `kernel.cl` which contains the OpenCL/CUDA kernel that is compiled and executed on the device. Section 5 contains instructions to execute a simple matrix element calculation (referred to as Hello World) on a valid device using OpenCL or CUDA and Python.

2 Programs Needed

Along with a copy of MadGraph 5 v. 1.5.14 with the OpenCL/CUDA extension the following programs are needed for the use of the outputted code using a Python, PyOpenCL and PyCUDA interface. Appendix B contains the console commands for downloading and installing all of these programs (except gcc).

AMD APP SDK / OpenCL

Either AMD APP SDK / OpenCL or CUDA is needed in order to execute the MadGraph outputted code. OpenCL code can be compiled and run on any GPU or on multiple CPU's working in parallel. AMD APP SDK v. 2.9 and OpenCL v. 1.2 were used for the Hello World.
Available from: `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`
`http://www.khronos.org/opencl/`

CUDA

Either AMD APP SDK / OpenCL or CUDA is needed in order to execute the MadGraph outputted code. CUDA is restricted to use on NVidia manufactured GPU's. CUDA v. beta 2.0 or newer is

needed. CUDA v. 4.2.9 was used for the Hello World.
Available from: `https://developer.nvidia.com/cuda-zone`

Python 2.7 / NumPy

A working Python Installation, v. 2.4 or newer. Untested on v. 3 and higher. Python v. 2.7.6 (Anaconda 2.0.0) was used for the Hello World.
Available from: `https://www.python.org/`
Anaconda distribution available from: `http://continuum.io/downloads`
PyOpenCL and PyCUDA are designed to work with NumPy, Python's array package, v. 1.0.4 or newer. NumPy v. 1.8.1 was used for the Hello World.
Available from: `http://www.numpy.org/`

ROOT / PyROOT

A working installation of ROOT and the Python package, PyROOT. ROOT v. 5.34/18 was used for the Hello World.
Available from: `http://root.cern.ch/drupal/content/installing-root-source`
`http://root.cern.ch/drupal/content/pyroot`

PyOpenCL

A working installation of the Python wrapper for OpenCL, PyOpenCL. Only required if using AMD APP SDK / OpenCL. PyOpenCL v. 2014.1 was used for the Hello World.
Available from `http://mathema.tician.de/software/pyopencl/`

PyCUDA

A working installation of the Python wrapper for CUDA, PyCUDA. Only required if using CUDA. PyCUDA v. 2013.1.1 was used for the Hello World.
Available from `http://mathema.tician.de/software/pycuda/`

C++ Compiler

The C++ compiler GCC v. 4.x along with the fortran compiler, Gfortran. GCC v. 4.8.1 was used for the Hello World.

3 MadGraph 5 OpenCL/CUDA Output

To create the OpenCL/CUDA output from MadGraph 5 begin by opening the MadGraph 5 interface:

```
$ path_to_MG5/bin/mg5
```

Generate a process:

```
mg5> generate p p > t t~
```

Create the OpenCL/CUDA stand alone output with the directory name `pptt_output`:

```
mg5> output standalone_ocl pptt_output
```

In the newly created directory there should be three directories, `kernel`, `lib`, `src`, and a python file, `kernel.py`. The files created provide the code necessary to calculate a matrix element using an OpenCL or CUDA kernel executed on a valid device. This calculation can also be run on a host side CPU using standard C++. The compilation method is determined by pre-compiler flags.

Provided below is a quick summary of the OpenCL/CUDA output and their function in calculating the matrix element.

`cards/`

Contains params.dat, the listing of the physical parameters for the model used by MadGraph 5 at the time the process was generated.

`lib/`

Contains the shared object library for the model being used. Additional shared libraries will be placed in this folder.

`src/`

Contains the files holding the functions that will be called by the kernel. These include the functions necessary to create/access the parameters of the model being used, and functions for the evaluation of the PDF that will be used to calculate the matrix element. As well, the HELAS functions for the matrix element calculators are present here.

`src/matcommon`

Contains the preprocessor definitions used through the files that allow the OpenCL/CUDA output to be executed using OpenCL or CUDA on a valid device or as a C++ function able to be executed on a host side CPU. These different modes are set at compilation time of the kernel. The flag, `_CL_CUDA_READY_` , along with either of the flags `_OPENCL_` or `_CUDA_` will set the compilation and run mode to OpenCL or CUDA respectively. Without the flag `_CL_CUDA_READY_` the kernel can be run on a host side CPU. These flags are set at compilation time in the `kernel.py` file.

`src/cmplx`

A complex number type is necessary for the evaluation of the matrix element. However, OpenCL does not have support for a complex number data type. `cmplx` creates a complex number data type, `a_cmplx_t`, that can be used by OpenCL while running on a device. Although CUDA does have a dedicated complex number data type, `a_cmplx_t` has some methods that facilitate seamless treatment of expressions on host and device. `src/helas_sm` .

`src/helas_sm`

Contains the HELAS functions used by the matrix element calculators. Modified to allow for OpenCL and CUDA compatibility.

`src/parameters_sm`

Contains the necessary code to create all the parameters of the model used by the matrix element calculators. Modified to allow for OpenCL and CUDA compatibility.

`src/pdf`

Contains the necessary code for the evaluation of PDF's with unified interface for LHAPDF and CTEQ. Also provides function for PDF evaluation on the device using bilinear interpolation of a (q, x) grid loaded in device memory.

`kernel/`

Contains the Opencl/CUDA kernel that will be compiled and executed by the selected device. This directory also contains the matrix element calculators for each subprocess present in the generated MadGraph process.

`kernel/kernel.cl`

The code that will be compiled and executed by the device. The kernel created by the extension is only a working template and may need to be altered. See the Section 4 for an overview of the kernel.

`kernel.py`

This is a python wrapper that uses PyOpenCL and PyCUDA to simplify the overhead needed to load data to a device and compile the kernel. It is here that build options for the compilation of the kernel are set.

Process Calculators

These calculators, one for each subprocess of the generated MadGraph process, calculate the matrix element given the kinematic variables of the external lines. These folders are named based on the subprocess they will calculate. For example in `pptt_output/kernel/` the calculators have the names `P0_Sigma_sm_gg_ttx` for $gg \to t\bar{t}$ and `P0_Sigma_sm_uux_ttx` for, not just $u\bar{u} \to t\bar{t}$, but all $q\bar{q} \to t\bar{t}$.

4 Overview of the Kernel

`kernel.cl` contains the function `eval` that is the main stage of the matrix element calculation to be executed. `eval` takes in arguments from the host and passes the kinematic variables of an event to the included functions and returns the final matrix element result. It is in this file that the OpenCL/CUDA output will be modified for any given project.

Presented here is a line by line summary of `kernel.cl` and the `eval` function so any user may more easily understand what is being done and how to adapt it to the user's needs. Note: many functions, data types and values are defined in matcommon.h. Consult this file for a full understanding of the code.

Lines 1-36

- a few useful preprocessor definitions and macros
- included files that contain the necessary functions
- `_nexternel` is the number of external lines in the MadGraph process generated
- `_nkin` is the number of kinematic variables for these external lines and the number of variables expected by the matrix element calculator(s)

Lines 39-44

- the 6 arguments that the `eval` fuctions takes and their intended use:
  - `xi`: kinematic variables that are not intended to be integrated in a Matrix Element Method implementation
  - `xp`: kinematic variables that are intended to be integrated in a Matrix Element Method implementation
  - `y`: results that will be returned to the host
  - `pdf_grd`: the 2 dimensional grid (x, Q^2) created to evaluate the PDF on the device
  - `pdf_bnds`: the boundaries of `pdf_grd`, 4 components $(x_{low}, x_{high}, Q^2_{low}, Q^2_{high})$
  - `pars`: the external parameters in the model

Line 46

- `idx`, defined in `src/matcommon.h`, gets the index of the device's compute unit that is evaluating the code. It can then be used as an array index to have each compute unit using a different set of data from the input arrays to have parallel evaluation.

Lines 48-52

- takes the external parameters given as an argument (`pars`) and creates all the physical parameters used by the model using the functions in `parameters_sm`. Parameters can be queried by `par_st.parameter_name`

Line 63

- `x[_nexternal][4]` is the array of kinematic variables of all the external lines that will be passed to the process calculators. The order of these lines is set when the process is generated. View the diagrams of the process generated to see the proper order. The order can also be seen as the names of the directories that hold the process calculators for each subprocess. For example, `P0_Sigma_sm_uux_ttx` expects the order of kinematic variables to be $u\bar{u}t\bar{t}$. The order of the kinematic variables is energy, x momentum, y momentum, z momentum.
- It is at this point that the code will be modified to suit the needs of each individual project. The inputs can be modified by additional functions and the kinematics of the external lines set in `x[][]`.

Lines 66-69

- the components of `x[][]` are set, in this template the values are set from the `xi` argument
- Q^2, the momentum transferred by the colliding patrons
- initializes the amplitude that will have the result of the process calculators

Lines 72-83

- the `x[][]` array is passed to each subprocess calculator using the function `set_momenta(x)` and the matrix element is calculated using the function `sigma_kin()` and stored in the array `matrix_element[]`
- the calculators are then queried to obtain the matrix element for each subprocess and multiplied by the two PDF factors and added to the total amplitude

Lines 84-87

- creates the Lorentz invariant phase space factor
- the result array, `y`, is set to the final matrix element value. The component of `y` is set based on the index, `idx`, thus each component can be set by a different compute unit for parallel computation.

5 OpenCL/CUDA Matrix Element Evaluation (Hello World)

Presented here are the steps that can be used to compile and run the template kernel using OpenCL or CUDA. This Hello World continues from the output generated in Section 3.

Step 1

Shared libraries must be made from the files in `src/`. `libmodel_sm.so` should be present in the `lib/` directory. If not:

```
$ cd pptt_output/src/
$ make
```

Create the remainder of the shared libraries:

```
$ cd pptt_output/src/ #if not there already
$ . compile.sh
```

Step 2

The PDF data being used is CTEQ 10. The file needed for this Hello World can be downloaded from `hep.pa.msu.edu/cteq/public/ct10_2010/pds/ct10.pds.zip` Place the file `ct10.00.pds` (or desired pdf data file) in `pptt_output/pdfs`

Step 3

Next we will create a Python script that will interface with the Python wrapper, `kernel.py`, to create the inputs for the kernel evaluation on the device and receive the final results. For this Hello World the script is named `analysis.py`. Shown in Appendix 1 is an example script that will do the job. The shown script also contains functionality to run the `kernel.cl eval` function on a host side CPU using C++. In the example shown, lines 16-35 are used to load the collision data of an event into a one dimensional array that will be passed to the kernel. The method shown requires an additional ExRootAnalysis library present in the `lib/` folder. This library can be obtained by installing ExRootAnalysis package through MadGraph 5:

```
mg5> install ExRootAnalysis
```

libExRootAnalysis will then be present in `path_to_mg5/ExRootAnalysis/lib/`. The collision data is loaded in from `input_data.root`. This file can be generated by MadGraph 5.

```
mg5> generate p p > t t~
mg5> launch
```

Edit the parameter and run card as desired. For this Hello World only one event is needed. Uncompress and convert the output event file.

```
$ gunzip PROC_sm_0/Events/run_01/unweighted_events.lhe.gz
     #adjust to proper output directory
$ path_to_MG5/ExRootAnalysis/ExRootLHEFConverter  PROC_sm_0/Events/run_01/
        unweighted_events.lhe  pptt_output/input_data.root
        #adjust to proper path names
```

Any other method that creates a one dimensional array of the kinematic variables of the external lines can be substituted in place of the method shown. The shown script can run the kernel using OpenCL or CUDA. This is set by setting the `mode` argument at line 49 to `opencl` or `cuda`. This script also contains functionality to run the `kernel.cl eval` function on a host side CPU using C++. This requires a copy of `kernel.cl` named `kernel.cc` be present in the `kernel/` directory. A symlink can be used to ensure the device and the CPU are running the same program.

```
$ cd pptt_output/kernel
$ ln -s kernel.cl kernel.cc
```

This controlling Python script will most likely be where integration functionality will be implemented as needed by the project.

Step 4

Compile and run the kernel:

```
$ cd pptt_output/
$ python analysis.py
```

For OpenCL the device used is set with the environment variable PYOPENCL_CTX or with a prompt at runtime. For CUDA the device used is set with the environment variable CUDA_DEVICE .

```
$ PYOPENCL_CTX=1 python analysis.py
```

or

```
$ CUDA_DEVICE=1 python analysis.py
```

As well, at the beginning of kernel.py are options to have the device selecton hard coded in.

To see available valid devices present run the following commands in a Python session. Using OpenCL:

```
>>> import pyopencl as cl
>>> for platform in cl.get_platforms():
...     for i, device in enumerate(platform.get_devices()):
...             print "Platform name:", platform.name
...             print "Device name:", device.name
...             print "Device type:", cl.device_type.to_string(device.type)
...             print "If Platform:", platform.name, "is used set
                        PYOPENCL_CTX =", i ,"to use", device.name
```

Using CUDA

```
>>> import pycuda.autoinit
>>> import pycuda.driver as cu
>>> for devicenum in range(cu.Device.count()):
...     device=cu.Device(devicenum)
...     print "To use",device.name(),"set CUDA_DEVICE =",devicenum
...     print "Set CUDA_DEVICE=",devicenum,"to use",device.name()
```

6 Further Steps

The OpenCL/CUDA output creates a basic template for a OpenCL or CUDA matrix element calculator and should not be considered a physically correct calculator straight out of the box. The kernel.cl and the analysis.py files must be changed and tailored depending on the process(es) and the project being investigated. A few changes that may be necessary are listed below.

- As mentioned before the order of the kinematics variables of the external lines must match what is expected by the calculator. Look to the generated diagrams or the calculator's directories for the proper order.
- The flavours of the incoming particles used in the PDF factor are by default set to 0 (gluons). Consult src/pdf.h line 48 for a list of flavours and their appropriate code.
- For speed on GPU devices the default data type used throughout the calculation is single floating point precision (32-bit). This can be changed in matcommon.h by changing the definition of a_float_t. A similar definition is present in kernel.py for the Python side of the code.

- Many values and functions used throughout the program are defined in `matcommon.h`. This file should be consulted and/or altered to suit the needs of the project at hand.
- Additional optimization and build options are available for the compilation of the kernel. Look to the appropriate (py)OpenCL and (py)CUDA documentation.

A Analysis.py

```
#control script to run Matrix Evaluation on a device using OpenCL/CUDA
#interfaced through kernel.py

import ROOT
import numpy as np
import kernel

#set # of external lines and the # of kinematic values for these lines
nexternal = 4   ##created for the pp->tt~ process
ndim = 4*nexternal
nevts = 1

#load collision data to a one dimensional np.array.
#The method shown here is reading the data from a .root file created by
#        MadGraph

#an extra library need to load the data in this method
ROOT.gSystem.Load('lib/libExRootAnalysis.so')

f = ROOT.TFile("input_data.root",'read')
t = f.Get('LHEF')
nevts = min(nevts,t.GetEntries())
nparticles = nexternal
event_data = np.zeros((nevts*nparticles*4))

for ievt in range(nevts):
        t.GetEntry(ievt)
        n = t.Particle.GetEntries()
        particles = [ t.Particle[0], t.Particle[1] ] + [ t.Particle[ip] for ip
                in range(n-2,n) ]
        for ip, p in enumerate(particles):
                event_data[ievt*nparticles*4 + ip*4 + 0] = p.E
            event_data[ievt*nparticles*4 + ip*4 + 1] = p.Px
            event_data[ievt*nparticles*4 + ip*4 + 2] = p.Py
            event_data[ievt*nparticles*4 + ip*4 + 3] = p.Pz
f.Close()

#Load shared library of src/ files to ROOT
ROOT.gSystem.Load('lib/libme.so')
```

```
#read in the parameters of the model using the SLHAReader
card_reader = ROOT.SLHAReader('cards/params.dat')
pobj = ROOT.parameters_sm()
pobj.set_independent_parameters(card_reader)
pobj.set_independent_couplings();
pobj.set_dependent_parameters();
pobj.set_dependent_couplings();

#initialize the kernel.py kernel class
#set mode = 'cuda' or mode = 'pyopencl'
me_obj = kernel.kernel(nexternal, ndim, pobj, pdfsetn = 'CT10', kernelfn =
        'kernel/kernel.cl', mode = 'opencl')

#create the result array
result = np.zeros(nevts,dtype=kernel.a_float_t)

#set the kinematic values for the event
me_obj.set_momenta((event_data[0*ndim:(0+1)*ndim]).astype(kernel.a_float_t))

#evaluate the matrix element
result = me_obj.eval(None)

##-----------------------------------------------------##
## additional functionality to run on a host side CPU ##

#this requires a copy of kernel/kernel.cl named kernel/kernel.cc

#Load the cpp version of the file to ROOT and set the proper directories to be
#        included
ROOT.gSystem.AddIncludePath(' -Isrc/ ')
ROOT.gROOT.ProcessLine('.L kernel/kernel.cc+')

cpu_result = np.zeros(1,dtype = kernel.a_float_t)

xp = np.zeros(ndim, dtype = kernel.a_float_t) #doesn't end up being used in eval

ROOT.eval((event_data[0*me_obj.nexternal*4:(0+1)*me_obj.nexternal*4]).astype(
        kernel.a_float_t), xp, cpu_result, me_obj.pdf_data, me_obj.pdf_bnds,
        me_obj.pars)

#print the device result
print 'The result calculated on', me_obj.ctx_name, 'is: '
print result

#print the CPU result
print 'The result calculed on the CPU is:'
print cpu_result
```

B Installing Programs

The console commands necessary to install all the programs listed in Section 2 and run the Hello World on 64-bit (x86_64) Ubuntu v13.04 using GNU bash, version 4.2.45(1)-release with gcc assumed installed.

Install Anaconda v2.0.1:

```
$ apt-get install wget #if wget is not already installed
$ wget 09c8d0b2229f813c1b93-c95ac804525aac4b6dba79b00b39d1d3.r79.cf1.rackcdn.com/
        Anaconda-2.0.1-Linux-x86_64.sh #or download manually
$ bash Anaconda-2.0.1-Linux-x86.sh
$export PATH = ~/anaconda/bin:$PATH #add to .bashrc file to avoid having to use
        this command at the beginning of every new console session
$ #check that it was installed properly
$ python
>>> import numpy as np
>>> quit()
```

Install ROOT & PyROOT v5.34/20:

```
$ wget ftp://root.cern.ch/root/root_v5.34.20.source.tar.gz
$ gzip -dc root_v5.34.20.source.tar.gz | tar -xf -
$ cd root
$ ./configure --enable-python --with-python-incdir= ~/anaconda/include
         --with-python-libdir= ~/anaconda/lib
         #alter include and library directories as necessary
$ gmake
$ . bin/thisroot.sh
$ export LD_LIBRARY_PATH=$ROOTSYS/lib:$PYTHONDIR/lib:$LD_LIBRARY_PATH
$ export PYTHONPATH=$ROOTSYS/lib:$PYTHONPATH
$ #check that it was installed properly
$ python
>>> import ROOT
>>> quit()
$ cd ~
```

Install AMD APP SDK v2.9 OpenCL & PyOpenCL 2014.1:
download AMD-APP-SDK-v2.9-lnx64.tgz from `http://developer.amd.com/tools-and-sdks/opencl-zone/amd-accelerated-parallel-processing-app-sdk/`
accept the licensing agreement and download

```
$ tar zxvf AMD-APP-SDK-v2.9-lnx32.tgz
$ sh ./Install-AMD-APP.sh
$ ln -sf /opt/AMDAPP/include/CL /usr/include
$ ln -sf /opt/AMDAPP/lib/x86/* /usr/lib/
$ ldconfig
$ pip install pyopencl --user
$ #check that it was installed properly
$ python
>>> import pyopencl
>>> quit()
```

Install CUDA v6.0 & PyCUDA v2013.1.1:
follow the appropriate installation guide in `http://docs.nvidia.com/cuda/cuda-getting-started-guide-for-linux/#axzz3Ara9aZCp`
you can find you linux distribution with

```
$ cat /etc/*release
```

for an installation on Ubuntu v13.04

```
$ wget http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1304/
        x86_64/cuda-repo-ubuntu1304_6.0-37_amd64.deb
$ sudo dpkg -i cuda-repo-ubuntu1304_6.0-37_amd64.deb
$ sudo apt-get update
$ sudo apt-get install cuda
$ export PATH=/usr/local/cuda-6.0/bin:$PATH
$ export LD_LIBRARY_PATH=/usr/local/cuda-6.0/lib64:$LD_LIBRARY_PATH
$ pip install pycuda --user
$ #check the installation
$ python
>>> import pycuda.autoinit
>>> quit()
```

Install MadGraph v5.14 with the OpenCL/CUDA mod:

```
$ wget <<modded MadGraph url>>
$ gunzip/tar/gzip the file
$ MadGraph5_v1_5_14_mod/bin/mg5
mg5> install ExRootAnalysis
mg5> generate p p > t t~
mg5> launch #edit run/param card
mg5> generate p p > t t~
mg5> output standalone_ocl pptt_output
mg5> quit
$ unzip PROC_sm_0/Events/run_01/unweighted_events.lhe.gz
$ MadGraph5_v1_5_14_mod/ExRootAnalysis/ExRootLHEFConverter PROC_sm_0/
        Events/run_01/unweighted_events.lhe pptt_output/input_data.root
$ #copy analysis.py into pptt_output, edit compilation mode to opencl or cuda
$ cd pptt_output/src/
$ . compile.sh
$ cd ..
$ cp ../MadGraph5_v1_5_14_mod/ExRootAnalysis/lib/libExRootAnalysis.so lib/
$ mkdir pdfs/
$ cd pdfs/
$ wget hep.pa.msu.edu/cteq/public/ct10_2010/pds/ct10.pds.zip
$ unzip ct10.pds.zip
$ cd ../kernel/
$ ln -s kernel.cl kernel.cc
$ python analysis.py #select device at runtime or by altering code or environment
        variables
```

References

[1] J. E. Stone, D. Gohara, G. Shi, Opencl: A parallel programming standard for heterogeneous computing systems, IEEE Des. Test 12 (3) (2010) 6673. doi:10.1109/MCSE.2010.69.

[2] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with cuda, Queue 6 (2) (2008) 4053. doi:10.1145/1365490.1365500.

[3] A. Klöckner, N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, A. Fasih, PyCUDA and PyOpenCL: A Scripting-Based Approach to GPU Run-Time Code Generation, Parallel Computing 38 (3) (2012) 157174. doi:10.1016/j.parco.2011.09.001.

[4] J. Alwall, M. Herquet, F. Maltoni, O. Mattelaer, T. Stelzer, MadGraph 5 : Going Beyond, JHEP 1106 (2011) 128. arXiv:1106.0522, doi:10.1007/JHEP06(2011)128.

[5] K. Hagiwara, J. Kanzaki, Q. Li, N. Okamura, T. Stelzer, Fast computation of MadGraph amplitudes on graphics processing unit (GPU), European Physical Journal C 73 (2013) 2608. arXiv:1305.0708, doi:10.1140/epjc/s10052-013-2608-2.

[6] D. Schouten, A. DeAbreu, B. Stelzer, Accelerated Matrix Element Method with Parallel Computing, 2014.

[7] K. Kondo, Dynamical Likelihood Method for Reconstruction of Events with Missing Momentum. I. Method and Toy Models, J. Phys. Soc. Jap. 57 (12) (1988) 41264140. doi:10.1143/JPSJ.57.4126.

[8] F. Fiedler, A. Grohsjean, P. Haefner, and P. Schieferdecker, The Matrix Element Method and its Application to Measurements of the Top Quark Mass, NIM A 624 (1) (2010) 203218. doi:10.1016/j.nima.2010.09.024.

[9] A. Grohsjean, Measurement of the Top Quark Mass in the Dilepton Final State Using the Matrix Element Method, 2010. doi:10.1007/978-3-642-14070-9 5.

[10] H. Murayama, I. Watanabe, K. Hagiwara, HELAS: HELicity amplitude subroutines for Feynman diagram evaluations.

[11] P. de Aquino, W. Link, F. Maltoni, O. Mattelaer, T. Stelzer, ALOHA: Automatic Libraries Of Helicity Amplitudes for Feynman Diagram Computations, Comput.Phys.Commun. 183 (2012) 22542263. arXiv:1108.2041, doi:10.1016/j.cpc.2012.05.004.

[12] M. Whalley, D. Bourilkov, R. Group, The Les Houches accord PDFs (LHAPDF) and LHAGLUE arXiv:hep-ph/0508110.

[13] Lai, Hung-Liang and Guzzi, Marco and Huston, Joey and Li, Zhao and Nadolsky, Pavel M. and others, New parton distributions for collider physics, Phys.Rev. D82 (2010) 074024. arXiv:1007.2241, doi:10.1103/PhysRevD.82.074024.