Comparing Sycl & Kokkos with MadGraph

Taylor Childers, Nathan Nichols (ANL)

Metrics To Consider

https://arxiv.org/abs/2203.09945
Sheets

- Ease of learning for experts and novices
- Ease of code conversion
  - From CPU code to Accelerator (GPU, etc.) code
  - From low level (CUDA, etc.) to higher level portability code
  - From one portability framework to another
- Impact on other existing code
  - Extent of modifications to existing code: does it take over main(), does it affect the threading or execution model, etc.
  - Extent of modifications to Event Data Model (EDM): data transfer and access across different memory space, etc.
- Impact on existing tool chain and build infrastructure
  - Extent of modifications to build rules / system
  - Do we need to recompile the entire software stack?
  - CMake or make changes/integration
- Hardware mapping
  - Is the technology working on all current hardware architectures?
  - Support for new hardware features and new architectures
- Feature availability
  - Reductions, kernel chaining, callbacks, etc
  - Concurrent kernel execution – Support for interfacing to optimized math-heavy libraries (FFTs, etc.)
- Ease of Debugging
  - How easy is it to debug implementations of code in the technologies?
- Address needs of all types of workflows
  - Scaling with # of kernels / application
  - Scaling with # of developers
  - Support for users by portability technology developers
- Long-term sustainability and code stability
  - Support model of technologies, stability of implementation if underlying libraries (CUDA) change
  - CUDA is going to be around for a long time, what about the portability solutions?
  - Long term support for technologies by vendors
- Compilation time
  - Separate builds for different architectures?
  - Compatibility with experiment's software distribution strategies
- Performance: CPU and GPU
  - Does the portable code version (CPU and GPU uses same code) degrade the CPU performance or use more memory?
- Aesthetics
  - compatibility with C++ standards
- Interoperability
  - Can you mix portability technologies in the same application?
  - How are external packages treated if they are imported into experiment software stacks and use different portability technologies? (CMSSW [18, 19, 20, 21, 22] is using Kokkos, but Geant [23, 24, 25] is using Alpaka)
  - Interaction with existing thread pool on CPU/GPU back ends?

Argonne NATIONAL LABORATORY

SYCL

```cpp
#include <iostream>
#include <sycl/sycl.hpp>

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  sycl::queue my_queue;

  // create and allocate device objects
  float *d_a = sycl::malloc_device<float>(4, my_queue);
  float *d_b = sycl::malloc_device<float>(4, my_queue);
  float *d_c = sycl::malloc_device<float>(4, my_queue);

  // copy from host to device
  my_queue.memcpy(d_a, h_a, 4*sizeof(float));
  my_queue.memcpy(d_b, h_b, 4*sizeof(float));
  // wait for asynchronous copies to complete
  my_queue.wait();

  my_queue.parallel_for(4, [=](sycl::id<1> idx) {
    // perform vector addition according to own
    // rank number starting at 0
    d_c[idx] = d_a[idx] + d_b[idx];
  });
  my_queue.wait(); // wait for kernel to complete

  // copy results from device to host
  my_queue.memcpy(h_c, d_c, 4*sizeof(float)).wait();

  // free memory
  sycl::free(d_a, my_queue);
  sycl::free(d_b, my_queue);
  sycl::free(d_c, my_queue);

  return 0;
}
```

Kokkos

```cpp
#include <iostream>
#include <Kokkos_Core.hpp>

int main(int argc, char** argv) {
  float a[] = { 1.0, 2.0, 3.0, 4.0 };
  float b[] = { 4.0, 3.0, 2.0, 1.0 };

  Kokkos::initialize(argc,argv);
  {
    // wrap host data in View object
    Kokkos::View<float*,Kokkos::HostSpace> h_a(a,4);
    Kokkos::View<float*,Kokkos::HostSpace> h_b(b,4);

    // create device side Views
    Kokkos::View<float*> d_a("a",4);
    Kokkos::View<float*> d_b("b",4);
    Kokkos::View<float*> d_c("c",4);

    // copy from host to device
    Kokkos::deep_copy(d_a,h_a);
    Kokkos::deep_copy(d_b,h_b);

    Kokkos::parallel_for( 4, KOKKOS_LAMBDA (const int& i) {
      d_c(i) = d_a(i) + d_b(i);
    });

    // create host side for output:
    auto h_c = Kokkos::create_mirror_view(d_c);
    Kokkos::deep_copy(h_c,d_c);
  }
  Kokkos::finalize();

  return 0;
}
```

CUDA

```cpp
#include <iostream>

__global__ void addition(float* a, float* b, float* c){
  int idx = blockIdx.x *blockDim.x + threadIdx.x;
  c[idx] = a[idx] + b[idx];
}

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  // create and allocate device objects
  float *d_a = 0;
  float *d_b = 0;
  float *d_c = 0;

  cudaMalloc(&d_a,4*sizeof(float));
  cudaMalloc(&d_b,4*sizeof(float));
  cudaMalloc(&d_c,4*sizeof(float));

  cudaMemcpy(d_a,h_a,4*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_b,h_b,4*sizeof(float),cudaMemcpyHostToDevice);

  addition<<<1,4>>>(d_a,d_b,d_c);

  cudaMemcpy(h_c,d_c,4*sizeof(float),cudaMemcpyDeviceToHost);

  // free memory
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
}
```

```
  A { 1, 2, 3, 4 }
+ B { 4, 3, 2, 1 }
------------------
= C { 5, 5, 5, 5 }
```


Argonne NATIONAL LABORATORY

Infrastructure

SYCL

```
#include <iostream>
#include <sycl/sycl.hpp>

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  sycl::queue my_queue;

  // create and allocate device objects
  float *d_a = sycl::malloc_device<float>(4, my_queue);
  float *d_b = sycl::malloc_device<float>(4, my_queue);
  float *d_c = sycl::malloc_device<float>(4, my_queue);

  // copy from host to device
  my_queue.memcpy(d_a, h_a, 4*sizeof(float));
  my_queue.memcpy(d_b, h_b, 4*sizeof(float));
  // wait for asynchronous copies to complete
  my_queue.wait();

  my_queue.parallel_for(4, [=](sycl::id<1> idx) {
    // perform vector addition according to own
    // rank number starting at 0
    d_c[idx] = d_a[idx] + d_b[idx];
  });
  my_queue.wait(); // wait for kernel to complete

  // copy results from device to host
  my_queue.memcpy(h_c, d_c, 4*sizeof(float)).wait();

  // free memory
  sycl::free(d_a, my_queue);
  sycl::free(d_b, my_queue);
  sycl::free(d_c, my_queue);

  return 0;
}
```

Kokkos

```
#include <iostream>
#include <Kokkos_Core.hpp>

int main(int argc, char** argv) {
   float a[] = { 1.0, 2.0, 3.0, 4.0 };
   float b[] = { 4.0, 3.0, 2.0, 1.0 };

   Kokkos::initialize(argc,argv);
   {
     // wrap host data in View object
     Kokkos::View<float*,Kokkos::HostSpace> h_a(a,4);
     Kokkos::View<float*,Kokkos::HostSpace> h_b(b,4);

     // create device side Views
     Kokkos::View<float*> d_a("a",4);
     Kokkos::View<float*> d_b("b",4);
     Kokkos::View<float*> d_c("c",4);

     // copy from host to device
     Kokkos::deep_copy(d_a,h_a);
     Kokkos::deep_copy(d_b,h_b);

     Kokkos::parallel_for( 4, KOKKOS_LAMBDA (const int& i) {
       d_c(i) = d_a(i) + d_b(i);
     });

     // create host side for output:
     auto h_c = Kokkos::create_mirror_view(d_c);
     Kokkos::deep_copy(h_c,d_c);
   }
   Kokkos::finalize();

   return 0;
}
```

```
  A { 1, 2, 3, 4 }
+ B { 4, 3, 2, 1 }
------------------
= C { 5, 5, 5, 5 }
```

CUDA

```
#include <iostream>

__global__ void addition(float* a, float* b, float* c){
  int idx = blockIdx.x *blockDim.x + threadIdx.x;
  c[idx] = a[idx] + b[idx];
}

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  // create and allocate device objects
  float *d_a = 0;
  float *d_b = 0;
  float *d_c = 0;

  cudaMalloc(&d_a,4*sizeof(float));
  cudaMalloc(&d_b,4*sizeof(float));
  cudaMalloc(&d_c,4*sizeof(float));

  cudaMemcpy(d_a,h_a,4*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_b,h_b,4*sizeof(float),cudaMemcpyHostToDevice);

  addition<<<1,4>>>(d_a,d_b,d_c);

  cudaMemcpy(h_c,d_c,4*sizeof(float),cudaMemcpyDeviceToHost);

  // free memory
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
}
```

Infrastructure

Data Management

SYCL

```
#include <iostream>
#include <sycl/sycl.hpp>

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  sycl::queue my_queue;

  // create and allocate device objects
  float *d_a = sycl::malloc_device<float>(4, my_queue);
  float *d_b = sycl::malloc_device<float>(4, my_queue);
  float *d_c = sycl::malloc_device<float>(4, my_queue);

  // copy from host to device
  my_queue.memcpy(d_a, h_a, 4*sizeof(float));
  my_queue.memcpy(d_b, h_b, 4*sizeof(float));
  // wait for asynchronous copies to complete
  my_queue.wait();

  my_queue.parallel_for(4, [=](sycl::id<1> idx) {
    // perform vector addition according to own
    // rank number starting at 0
    d_c[idx] = d_a[idx] + d_b[idx];
  });
  my_queue.wait(); // wait for kernel to complete

  // copy results from device to host
  my_queue.memcpy(h_c, d_c, 4*sizeof(float)).wait();

  // free memory
  sycl::free(d_a, my_queue);
  sycl::free(d_b, my_queue);
  sycl::free(d_c, my_queue);

  return 0;
}
```

Kokkos

```
#include <iostream>
#include <Kokkos_Core.hpp>

int main(int argc, char** argv) {
  float a[] = { 1.0, 2.0, 3.0, 4.0 };
  float b[] = { 4.0, 3.0, 2.0, 1.0 };

  Kokkos::initialize(argc,argv);
  {
    // wrap host data in View object
    Kokkos::View<float*,Kokkos::HostSpace> h_a(a,4);
    Kokkos::View<float*,Kokkos::HostSpace> h_b(b,4);

    // create device side Views
    Kokkos::View<float*> d_a("a",4);
    Kokkos::View<float*> d_b("b",4);
    Kokkos::View<float*> d_c("c",4);

    // copy from host to device
    Kokkos::deep_copy(d_a,h_a);
    Kokkos::deep_copy(d_b,h_b);

    Kokkos::parallel_for( 4, KOKKOS_LAMBDA (const int& i) {
      d_c(i) = d_a(i) + d_b(i);
    });

    // create host side for output:
    auto h_c = Kokkos::create_mirror_view(d_c);
    Kokkos::deep_copy(h_c,d_c);
  }
  Kokkos::finalize();

  return 0;
}
```

```
  A { 1, 2, 3, 4 }
+ B { 4, 3, 2, 1 }
------------------
= C { 5, 5, 5, 5 }
```

CUDA

```
#include <iostream>

__global__ void addition(float* a, float* b, float* c){
  int idx = blockIdx.x *blockDim.x + threadIdx.x;
  c[idx] = a[idx] + b[idx];
}

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  // create and allocate device objects
  float *d_a = 0;
  float *d_b = 0;
  float *d_c = 0;

  cudaMalloc(&d_a,4*sizeof(float));
  cudaMalloc(&d_b,4*sizeof(float));
  cudaMalloc(&d_c,4*sizeof(float));

  cudaMemcpy(d_a,h_a,4*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_b,h_b,4*sizeof(float),cudaMemcpyHostToDevice);

  addition<<<1,4>>>(d_a,d_b,d_c);

  cudaMemcpy(h_c,d_c,4*sizeof(float),cudaMemcpyDeviceToHost);

  // free memory
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
}
```

Infrastructure | Data Management | Algorithm Management

SYCL

```
#include <iostream>
#include <sycl/sycl.hpp>

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  sycl::queue my_queue;

  // create and allocate device objects
  float *d_a = sycl::malloc_device<float>(4, my_queue);
  float *d_b = sycl::malloc_device<float>(4, my_queue);
  float *d_c = sycl::malloc_device<float>(4, my_queue);

  // copy from host to device
  my_queue.memcpy(d_a, h_a, 4*sizeof(float));
  my_queue.memcpy(d_b, h_b, 4*sizeof(float));
  // wait for asynchronous copies to complete
  my_queue.wait();

  my_queue.parallel_for(4, [=](sycl::id<1> idx) {
    // perform vector addition according to own
    // rank number starting at 0
    d_c[idx] = d_a[idx] + d_b[idx];
  });
  my_queue.wait(); // wait for kernel to complete

  // copy results from device to host
  my_queue.memcpy(h_c, d_c, 4*sizeof(float)).wait();

  // free memory
  sycl::free(d_a, my_queue);
  sycl::free(d_b, my_queue);
  sycl::free(d_c, my_queue);

  return 0;
}
```

Kokkos

```
#include <iostream>
#include <Kokkos_Core.hpp>

int main(int argc, char** argv) {
  float a[] = { 1.0, 2.0, 3.0, 4.0 };
  float b[] = { 4.0, 3.0, 2.0, 1.0 };

  Kokkos::initialize(argc,argv);
  {
    // wrap host data in View object
    Kokkos::View<float*,Kokkos::HostSpace> h_a(a,4);
    Kokkos::View<float*,Kokkos::HostSpace> h_b(b,4);

    // create device side Views
    Kokkos::View<float*> d_a("a",4);
    Kokkos::View<float*> d_b("b",4);
    Kokkos::View<float*> d_c("c",4);

    // copy from host to device
    Kokkos::deep_copy(d_a,h_a);
    Kokkos::deep_copy(d_b,h_b);

    Kokkos::parallel_for( 4, KOKKOS_LAMBDA (const int& i) {
      d_c(i) = d_a(i) + d_b(i);
    });

    // create host side for output:
    auto h_c = Kokkos::create_mirror_view(d_c);
    Kokkos::deep_copy(h_c,d_c);
  }
  Kokkos::finalize();

  return 0;
}
```

```
  A { 1, 2, 3, 4 }
+ B { 4, 3, 2, 1 }
------------------
= C { 5, 5, 5, 5 }
```

CUDA

```
#include <iostream>

__global__ void addition(float* a, float* b, float* c){
  int idx = blockIdx.x *blockDim.x + threadIdx.x;
  c[idx] = a[idx] + b[idx];
}

int main(int, char**) {
  float h_a[4] { 1.0, 2.0, 3.0, 4.0 };
  float h_b[4] { 4.0, 3.0, 2.0, 1.0 };
  float h_c[4];

  // create and allocate device objects
  float *d_a = 0;
  float *d_b = 0;
  float *d_c = 0;

  cudaMalloc(&d_a,4*sizeof(float));
  cudaMalloc(&d_b,4*sizeof(float));
  cudaMalloc(&d_c,4*sizeof(float));

  cudaMemcpy(d_a,h_a,4*sizeof(float),cudaMemcpyHostToDevice);
  cudaMemcpy(d_b,h_b,4*sizeof(float),cudaMemcpyHostToDevice);

  addition<<<1,4>>>(d_a,d_b,d_c);

  cudaMemcpy(h_c,d_c,4*sizeof(float),cudaMemcpyDeviceToHost);

  // free memory
  cudaFree(d_a);
  cudaFree(d_b);
  cudaFree(d_c);

  return 0;
}
```

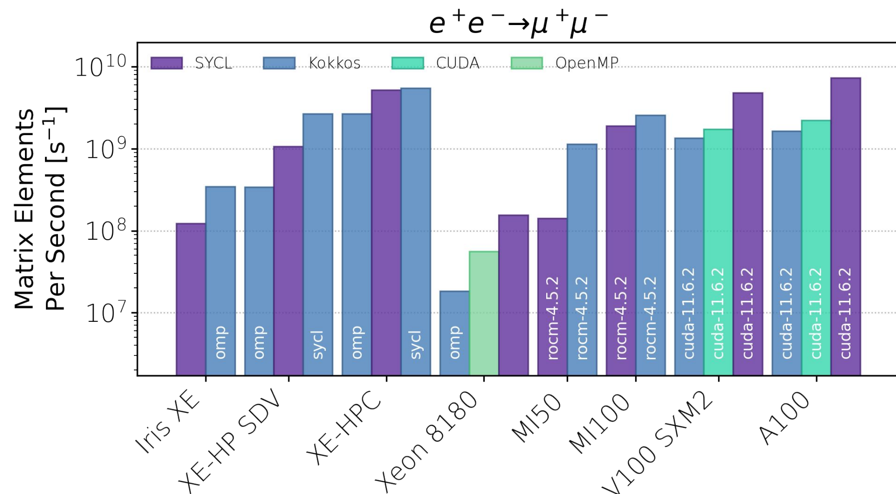
Some Metrics to Consider at This Point

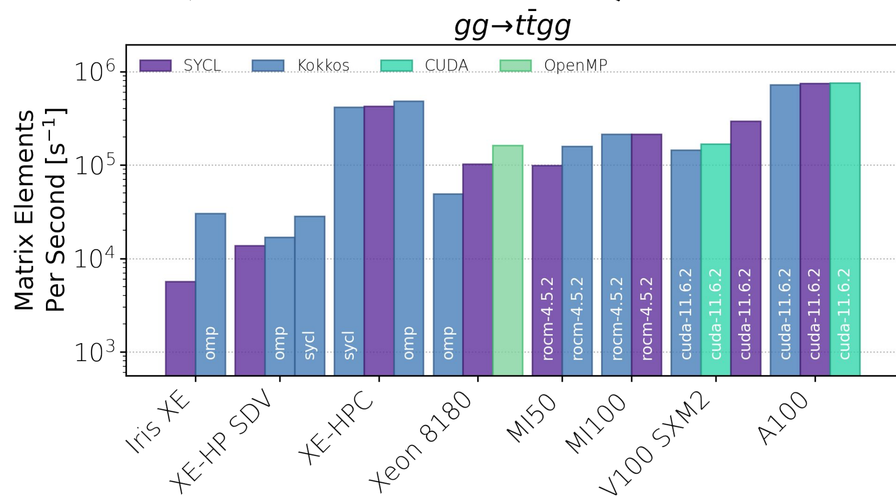
- Ease of learning for experts and novices
  - Compared to CUDA/C++, Kokkos and SYCL are both easy enough to pick up and use
  - Complexity comes with experience and understanding operational differences
- Ease of code conversion
  - If the code is already factored into parallel kernels, it is straightforward to convert this to SYCL or Kokkos
    - If not, it takes the same effort (maybe less) to convert to SYCL or Kokkos as CUDA
  - Memory management is still best done by hand, just as in CUDA
- Impact on other existing code
  - SYCL only requires a header to use
  - Kokkos requires a header and also the `` `initialize()` `` and `` `finalize()` `` calls
- Aesthetics
  - This is very personal, given the differences are negligible
  - Taylor: I like Kokkos aesthetically over SYCL but ONLY because of the context/queue stuff

Argonne NATIONAL LABORATORY

Performance: CPU and GPU

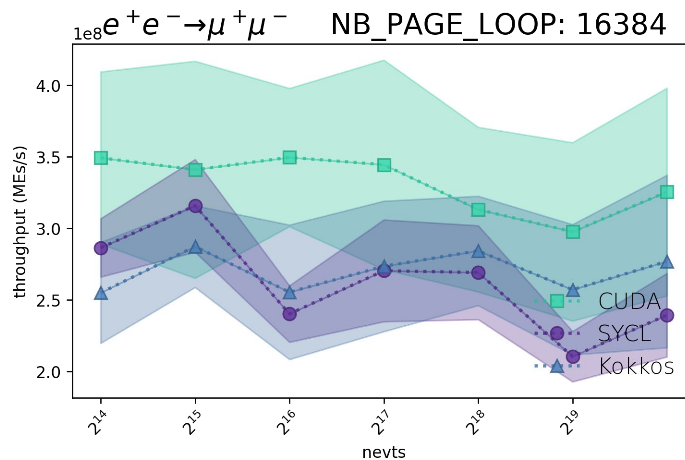
- Figure of Merit: Matrix Elements per second
- Measured using just the computational kernel, outside of MadEvent
- In the computationally intensive case, performance between SYCL/Kokkos/CUDA is comparable on all GPUs
- Some work to be done to understand differences on CPUs

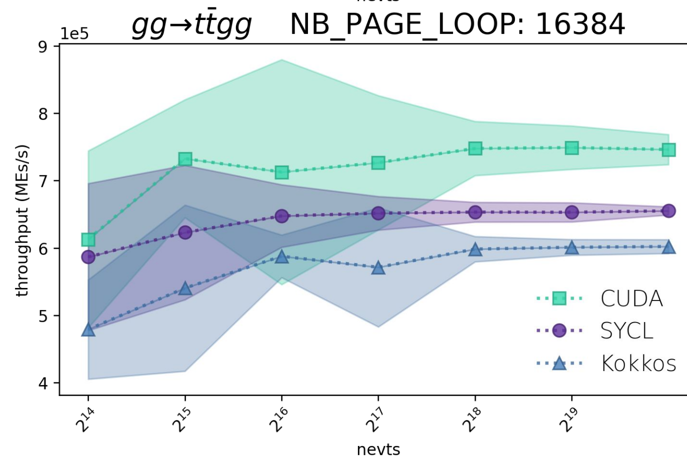




Performance: NVIDIA A100

- Figure of Merit: Matrix Elements per second
- Measured using just the computational kernel, outside of MadEvent + memory transfer of Matrix Elements
- `check.exe 64 256 2^<0|1|2|3|4|5>`
- Results are averaged over 10 runs with error bars showing 2 times the std. dev.
- The x-axis "`nevts`" is `64 * 256 * 2^<0|1|2|3|4|5> - 16384` except for the first point, which is `16384`
  - The data is represented in this strange way to better align the measured throughput with the MadEvent results on the next slide
- CUDA results from `master` branch
- SYCL/Kokkos results from `br_golden_epochX4` branch

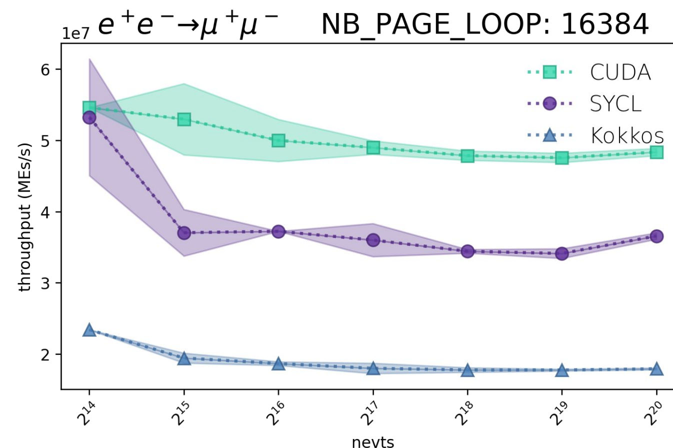


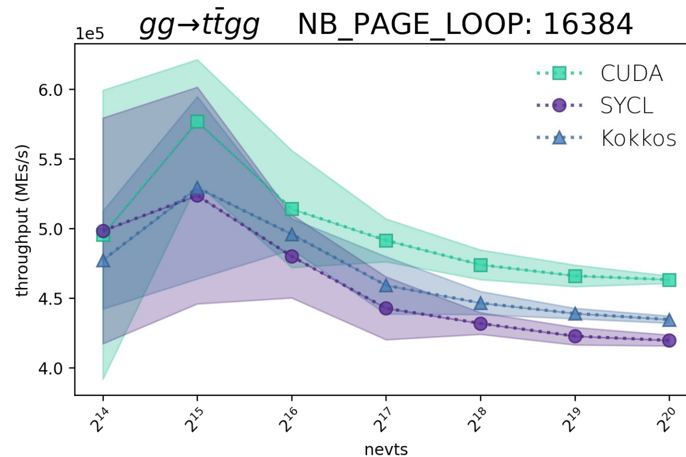


Argonne NATIONAL LABORATORY

Performance: NVIDIA A100

- Figure of Merit: Matrix Elements per second
- MadEvent results using stdout throughput from counters
- Results are averaged over 10 runs with error bars showing 2 times the std. Dev.
- `NB_PAGE_LOOP = 16384` results in 256 threads per block and 64 blocks per grid for each `gpu_sequence` call
- The x-axis `nevts` is the first argument to the "`Number of events and max and min iterations line`" of the input file
  - **Note:** The total number of matrix element calculations is `nevts + NB_PAGE_LOOP` unless `NB_PAGE_LOOP == nevts`, then it is `NB_PAGE_LOOP`
- Results additionally include transfer from host to device of the momenta and transpose of momenta for the CUDA/SYCL ports (compared to MadGraph SA)





Argonne NATIONAL LABORATORY

So Far…

SYCL or Kokkos or Either

https://arxiv.org/abs/2203.09945

Sheets

- Ease of learning for experts and novices
- Ease of code conversion
  - From CPU code to Accelerator (GPU, etc.) code
  - From low level (CUDA, etc.) to higher level portability code
  - From one portability framework to another
- Impact on other existing code
  - Extent of modifications to existing code: does it take over main(), does it affect the threading or execution model, etc.
  - Extent of modifications to Event Data Model (EDM): data transfer and access across different memory space, etc.
- Impact on existing tool chain and build infrastructure
  - Extent of modifications to build rules / system
  - Do we need to recompile the entire software stack?
  - CMake or make changes/integration
- Hardware mapping
  - Is the technology working on all current hardware architectures?
  - Support for new hardware features and new architectures
- Feature availability
  - Reductions, kernel chaining, callbacks, etc
  - Concurrent kernel execution – Support for interfacing to optimized math-heavy libraries (FFTs, etc.)
- Ease of Debugging
  - How easy is it to debug implementations of code in the technologies?
- Address needs of all types of workflows
  - Scaling with # of kernels / application
  - Scaling with # of developers
  - Support for users by portability technology developers
- Long-term sustainability and code stability
  - Support model of technologies, stability of implementation if underlying libraries (CUDA) change
  - CUDA is going to be around for a long time, what about the portability solutions?
  - Long term support for technologies by vendors
- Compilation time
  - Separate builds for different architectures?
  - Compatibility with experiment's software distribution strategies
- Performance: CPU and GPU
  - Does the portable code version (CPU and GPU uses same code) degrade the CPU performance or use more memory?
- Aesthetics
  - compatibility with C++ standards
- Interoperability
  - Can you mix portability technologies in the same application?
  - How are external packages treated if they are imported into experiment software stacks and use different portability technologies? (CMSSW [18, 19, 20, 21, 22] is using Kokkos, but Geant [23, 24, 25] is using Alpaka)
  - Interaction with existing thread pool on CPU/GPU back ends?

Building Software

- CUDA requires CUDA to be installed to build
  - Must use a mix of nvcc and gcc/g++
- SYCL requires a compiler with SYCL standard implemented
  - Intel's oneAPI DPC++ currently the most feature complete compiler
  - This may involve building your compiler if your system doesn't provide it.
  - Compiling for different gpu vendors requires additional compilation flags:
    - clang++ -fsycl -fsycl-targets=nvptx64-nvidia-cuda <…>
    - clang++ -fsycl -fsycl-targets=amdgcn-amd-amdhsa <…>
  - AOT compilation can be performed using the -Xsycl-target-backend flag.

- Kokkos is a C++ library that must be compiled for the architecture you want
  - Kokkos is built using CMake
    - Use CMake flags to set target device:
      - -DKokkos_ENABLE_OPENMP=On
      - -DKokkos_ENABLE_CUDA=On -DKokkos_ARCH_VOLTA70=On
      - -DKokkos_ENABLE_HIP=On -DKokkos_ARCH_VEGA908=On
  - If your package uses CMake, Kokkos will handle the compiler flags for you

Debugging Tools

- SYCL
  - Can use standard debugging tools (`gdb`, `valgrind`, `nvprof`, `vtune`, etc.)
- Kokkos
  - Comes with build-in debugging/monitoring via: https://github.com/kokkos/kokkos-tools
  - Easy timing of kernels and memory usage by adding a library via Env-var.
  - Can also use all the standard tools

Long-term sustainability and code stability

- This is the hardest to judge.
- SYCL:
  - Intel supported C++ Compiler Specification
  - Implemented as a C++ library.
  - Implementations for Intel, CUDA, and AMD are mostly supported by Intel software engineers (who own Codeplay now).
  - Worse Case Scenario: need to implement C++ code in intel/llvm/sycl library to support new features/hardware
    - These features may be implemented as extensions to the SYCL language (see Tensor Core support for example)
- Kokkos:
  - DOE Exascale Computing Project, supported by scientific community
  - Templated C++ library
  - Worse Case Scenario: need to implement C++ code in the Kokkos library for new features/hardware
- Generally:
  - Safe to say both solutions are going to be solid for at least 5 years, likely longer

SYCL or Kokkos or Either

https://arxiv.org/abs/2203.09945
Sheets

- Ease of learning for experts and novices
- Ease of code conversion
  - From CPU code to Accelerator (GPU, etc.) code
  - From low level (CUDA, etc.) to higher level portability code
  - From one portability framework to another
- Impact on other existing code
  - Extent of modifications to existing code: does it take over main(), does it affect the threading or execution model, etc.
  - Extent of modifications to Event Data Model (EDM): data transfer and access across different memory space, etc.
- Impact on existing tool chain and build infrastructure
  - Extent of modifications to build rules / system
  - Do we need to recompile the entire software stack?
  - CMake or make changes/integration
- Hardware mapping
  - Is the technology working on all current hardware architectures?
  - Support for new hardware features and new architectures
- Feature availability
  - Reductions, kernel chaining, callbacks, etc
  - Concurrent kernel execution – Support for interfacing to optimized math-heavy libraries (FFTs, etc.)
- Ease of Debugging
  - How easy is it to debug implementations of code in the technologies?
- Address needs of all types of workflows
  - Scaling with # of kernels / application
  - Scaling with # of developers
  - Support for users by portability technology developers
- Long-term sustainability and code stability
  - Support model of technologies, stability of implementation if underlying libraries (CUDA) change
  - CUDA is going to be around for a long time, what about the portability solutions?
  - Long term support for technologies by vendors
- Compilation time
  - Separate builds for different architectures?
  - Compatibility with experiment’s software distribution strategies
- Performance: CPU and GPU
  - Does the portable code version (CPU and GPU uses same code) degrade the CPU performance or use more memory?
- Aesthetics
  - compatibility with C++ standards
- Interoperability
  - Can you mix portability technologies in the same application?
  - How are external packages treated if they are imported into experiment software stacks and use different portability technologies? (CMSSW [18, 19, 20, 21, 22] is using Kokkos, but Geant [23, 24, 25] is using Alpaka)
  - Interaction with existing thread pool on CPU/GPU back ends?

Argonne NATIONAL LABORATORY