A COMPARATIVE ANALYSIS OF KOKKOS AND SYCL AS HETEROGENEOUS PARALLEL PROGRAMMING MODELS FOR C++ APPLICATIONS

Jeff Hammond, Michael Kinsner, James Brodman
Intel Corporation

IWOCL DHPCC++ 2019 (13 May 2019)

Notices and Disclaimers

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. . Performance varies depending on system configuration. **No computer system can be absolutely secure.** Check with your system manufacturer or retailer or learn more at intel.com/performance/datacenter.

Some results have been estimated or simulated using internal Intel analysis or architecture simulation or modeling, and provided to you for informational purposes. Any differences in your system hardware, software or configuration may affect your actual performance.

The cost reduction scenarios described are intended to enable you to get a better understanding of how the purchase of a given Intel based product, combined with a number of situation-specific variables, might affect future costs and savings. Circumstances will vary and there may be unaccounted-for costs related to the use and deployment of a given product. Nothing in this document should be interpreted as either a promise of or contract for a given level of costs or cost reduction.

Intel processors of the same SKU may vary in frequency or power as a result of natural variability in the production process.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice. Notice Revision #20110804.

Performance estimates were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown." Implementation of these updates may make these results inapplicable to your device or system. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to www.intel.com/benchmarks

Additional Disclaimer

I am not an official spokesman for any Intel products. I do not speak for my collaborators, whether they be inside or outside Intel.

I work on system pathfinding and workload analysis, not software products. I am not a developer of Intel software tools.

The PowerPoint C++ compiler is very lax about syntax – the code you see on the slides was derived from working code but has been modified for aesthetic appeal and may contain errors.

Hanlon's Razor (blame stupidity, not malice).

SYCL: Reactive and Proactive Motivation

Reactive to OpenCL Pros and Cons:

- OpenCL has a well-defined, portable execution model.
- OpenCL is prohibitively verbose for many application developers.
- OpenCL remains a C API and only recently supported C++ kernels.
- Disjoint host and kernel source code is awkward.

Proactive about Future C++:

- SYCL is based on purely modern C++ and should feel familiar to C++11 users.
- SYCL expected to run ahead of C++Next regarding heterogeneity and parallelism.
- Not held back by C99 or C++03 compatibility goals.

Kokkos: Motivation and Goals

- DOE wants/needs to run applications across a wide range of architectures:
  - CPU w/ big or small cores (e.g. Intel® Xeon® and Xeon Phi™)
  - GPU w/ and w/o unified memory (e.g. LLNL Sierra and ORNL Titan)
  - No common programming model across all platforms due to inconsistent vendor support for OpenMP* and OpenCL* ☹
- *Goal: write one implementation which:*
  - *compiles and runs on multiple architectures*
  - *obtains performant memory access patterns across architectures*
  - *can leverage architecture-specific features where possible.*

The goal text above is taken verbatim from the Kokkos SC15 tutorial.

DOE Exascale Systems (2021)

Kokkos will support both of these machines…

PARALLEL EXECUTION MODEL

Parallel execution – where to run by default?

Default:

cl::sycl::host_selector{}

"…selects a SYCL device based on an implementation defined heuristic. Must select a host device if no other suitable OpenCL device can be found."

Default:

Kokkos::initialize(..)

"…initializes the default execution space Kokkos::DefaultExecutionSpace."

The default depends on the Kokkos configuration. Kokkos documents the rule as:

ROCm > CUDA > OpenMP > Threads

(the priority of other cases is presumably documented in the source code)

Parallel execution – controlling where to run

Available devices:

- cl::sycl::host_selector{}
- cl::sycl::cpu_selector{}
- cl::sycl::gpu_selector{}
- cl::sycl::accelerator_selector{}

Available devices:

- Kokkos::Threads
- Kokkos::OpenMP
- Kokkos::OpenMPTarget
- Kokkos::Cuda
- Kokkos::ROCm
- Kokkos::HPX
- Kokkos::Qthreads

Parallel execution – example

```
queue q(cpu_selector{});

buffer<double,1> d_A { h_A, range<1>(n) };

q.submit([&](handler& h) {
  auto A = d_A. get_access<RW>(h);
  h.parallel_for<>(range<1>{n}, [=] (item<1> i){
    A[i]++;
  });
});
q.wait();
```

```
namespace K = Kokkos;

K::initialize(argc, argv);
typedef K::OpenMP Space;

typedef K::View<double*, Space> vector;
vector A("A", n);

auto range = K::RangePolicy<Space>(0,n);
K::parallel_for(range, KOKKOS_LAMBDA(int i) {
  A[i]++;
});

fence();
```

DATA MANAGEMENT MODEL

Array allocation parameters

SYCL buffer class parameters:

- Datatype
- Dimensions (1,2,3)
- Allocator

Accessors control access permissions.

Kokkos view class parameters:

- Datatype (built-in or struct of built-in)
- Dimensions (0,1,2,3,4,5,6,7,8)
- Space (optional)

Views can be const (assign from non-const view).

The Space may constrain the access rules (e.g. GPU cannot access host data unless UVM supported).

Accessing and moving data

SYCL data movement between host↔device(s) usually implicit based on DAG deps, but explicit available for tuning

Implicit:

- DAG dependence triggers data movement prior to kernel launch

Explicit:

- cl::sycl::handler::copy(..) requires source to be at least as big as target.
- cl::sycl::handler::update_host(..)

"Kokkos never performs a hidden deep copy."

Kokkos::deep_copy (out, in) but there are strict rules on what can be copied:

1. Identical memory layout and padding (likely different for host and device, e.g. OpenMP and CUDA)
2. HostMirror b = create_mirror(a)
3. HostMirror b = create_mirror_view(a)

2 always copies but 3 is a no-op when a is host memory.

COMPUTE PRIMITIVES

Parallel for and nested loops

```
// SYCL supports 1..3 dimensions

h.parallel_for<>(range<2>{n,n}, [=] (item<2> it) {
    id<2> ij{it[0],it[1]};
    id<2> ji{it[1],it[0]};
    B[ij] = A[ji];
});

h.parallel_for<>(range<2>{n,n}, [=] (item<2> it) {
    B[it[0] * n + it[1]] = A[it[1] * n + it[0]];
});
```

I haven't figured out which one I'm supposed to use, because the performance of the former has been much worse in some of my experiments…

```
// MDRP = MDRangePolicy

// MDRP supports 2 to 6 dimensions
auto policy = K::MDRP<K::Rank<2>>({0,0},{n,n},{t,t});

K::parallel_for(policy, KOKKOS_LAMBDA(int i, int j) {
    B(i,j) = A(j,i);
});
```

Kokkos supports tiling and access pattern (row or column major), but it's not clear how useful these are…

https://github.com/kokkos/kokkos/wiki/Kokkos::MDRangePolicy

Other parallel patterns

Like OpenCL* and CUDA* C/C++, SYCL assumes the user or a library implements patterns like reduce and scan.

Intel is working on a language extension…

Khronos SYCL parallel STL is a library solution:

```
namespace pstl = std::experimental::parallel;

cl::sycl::queue q;
pstl::sycl::sycl_execution_policy<..> snp(q);
int result = pstl::reduce(snp, v.begin(), v.end());
```

https://github.com/KhronosGroup/SyclParallelSTL

K::parallel_reduce – reductions with built-in and custom reduction operators.

K::parallel_scan – prefix sum.

Kokkos built-in reductions include everything that MPI_Reduce supports, even the dumb stuff (prod).

Example:

```
double out(0);
K::parallel_reduce(n, [=] (int i, double & tmp) {
        tmp += …;
}, out);
```

https://github.com/kokkos/kokkos/wiki/Data-Parallelism

NESTED PARALLELISM

This is where it all started for us ☺

Kokkos vs SYCL

Kokkos Name	SYCL Name
Thread team	Work-group
Thread league	Global range
Team scratch pad memory	Local memory

Kokkos Construct	SYCL Construct
parallel_for(TeamPolicy)	parallel_for_work_group(#wg, wg_size)
parallel_for(TeamThreadRange)	parallel_for_work_item (flex_range)
parallel_for(ThreadVectorRange) – WG or WI scope	parallel_for_sub_group
Barrier not implicit on ParFor(TeamThreadRange)	Implicit barrier at PFWI boundaries
single(PerTeam) – execute λ once per team	Code at PFWG scope
single(PerThread) – execute λ in single vec lane	Code at PFSG scope

Kokkos: nested parallelism

```
// TTR = TeamThreadRange
// TVR = ThreadVectorRange

typedef typename K::TeamPolicy<>::member_type tm;

struct foo {
  void operator() ( const tm& thread) const {
    int i = thread.league_rank();
    K::parallel_for(K::TTR(thread,jmax), [=] (const int& j) {
      K::parallel_for(K::TVR(thread,kmax), [=] (const int& k) {
        printf("foo %d %d %d\n", i, j, k);
      });
    });
  }
};

const K::TeamPolicy<> policy( imax , K::AUTO , 1);
K::parallel_for( policy , foo() );
```

```
foo 1 0 0
foo 1 0 1
foo 1 1 0
foo 1 1 1
foo 1 2 0
foo 1 2 1
foo 2 0 0
foo 2 0 1
foo 2 1 0
foo 2 1 1
foo 3 0 0
foo 3 0 1
foo 2 2 0
foo 2 2 1
foo 0 0 0
foo 0 0 1
foo 0 1 0
foo 0 1 1
foo 0 2 0
foo 0 2 1
foo 3 1 0
foo 3 1 1
foo 3 2 0
foo 3 2 1
```

```
int imax = 4;
int jmax = 3;
int kmax = 2;
```

SYCL: nested parallelism

```
q.submit([&](cl::sycl::handler& h) {
 h.parallel_for_work_group<class foo>(
                          cl::sycl::nd_range<1>(imax*jmax,jmax),
                          [=] (cl::sycl::group<1> g) {
     g.parallel_for_work_item( [=] (cl::sycl::h_item<1> i) {
         printf("foo g=%zu i=%zu\n", g.get_id(0), i.get_global_id());
     });
 });
});
```

```
foo g=0 i=1
foo g=0 i=2
foo g=0 i=0
foo g=1 i=4
foo g=1 i=5
foo g=1 i=3
foo g=2 i=7
foo g=2 i=6
foo g=2 i=8
foo g=3 i=10
foo g=3 i=11
foo g=3 i=9
```

```
int imax = 4;
int jmax = 3;
int kmax = 2;
```

SYCL: nested parallelism

```
using namespace cl::sycl;

  q.submit([&](handler& h) {
   h.parallel_for_work_group<class bar>(
              nd_range<2>({imax*kmax,jmax*kmax},{kmax,kmax}),
               [=] (group<2> g) {

     g.parallel_for_work_item( [=] (h_item<2> i) {
        printf("bar g[0]=%zu g[1]=%zu i[0]=%zu i[1]=%zu\n",
                     g.get_id(0), g.get_id(1),
                     i.get_global_id(0), i.get_global_id(1));
     });
   });
  });
```

```
bar g[0]=0 g[1]=0 i[0]=0 i[1]=1
bar g[0]=0 g[1]=0 i[0]=1 i[1]=0
bar g[0]=0 g[1]=0 i[0]=0 i[1]=0
bar g[0]=0 g[1]=0 i[0]=1 i[1]=1
bar g[0]=0 g[1]=1 i[0]=0 i[1]=3
bar g[0]=0 g[1]=1 i[0]=1 i[1]=2
bar g[0]=0 g[1]=1 i[0]=1 i[1]=3
bar g[0]=0 g[1]=1 i[0]=0 i[1]=2
bar g[0]=1 g[1]=0 i[0]=3 i[1]=0
bar g[0]=1 g[1]=0 i[0]=2 i[1]=0
bar g[0]=1 g[1]=0 i[0]=2 i[1]=1
bar g[0]=1 g[1]=0 i[0]=3 i[1]=1
bar g[0]=1 g[1]=1 i[0]=3 i[1]=2
bar g[0]=1 g[1]=1 i[0]=2 i[1]=2
bar g[0]=1 g[1]=1 i[0]=2 i[1]=3
bar g[0]=1 g[1]=1 i[0]=3 i[1]=3
bar g[0]=2 g[1]=0 i[0]=4 i[1]=0
bar g[0]=2 g[1]=0 i[0]=4 i[1]=1
bar g[0]=2 g[1]=0 i[0]=5 i[1]=0
bar g[0]=2 g[1]=0 i[0]=5 i[1]=1
bar g[0]=2 g[1]=1 i[0]=4 i[1]=3
bar g[0]=2 g[1]=1 i[0]=4 i[1]=2
bar g[0]=2 g[1]=1 i[0]=5 i[1]=3
bar g[0]=2 g[1]=1 i[0]=5 i[1]=2
bar g[0]=3 g[1]=0 i[0]=6 i[1]=1
bar g[0]=3 g[1]=0 i[0]=7 i[1]=0
bar g[0]=3 g[1]=0 i[0]=7 i[1]=1
bar g[0]=3 g[1]=0 i[0]=6 i[1]=0
bar g[0]=3 g[1]=1 i[0]=7 i[1]=2
bar g[0]=3 g[1]=1 i[0]=6 i[1]=3
bar g[0]=3 g[1]=1 i[0]=6 i[1]=2
bar g[0]=3 g[1]=1 i[0]=7 i[1]=3
```

```
int imax = 4;
int jmax = 2;
int kmax = 2;
```

CONCLUSIONS

- Application developers should be giving Kokkos a serious look if they want to support all three major HPC accelerator platforms.
- SYCL needs to learn from Kokkos:
  - Reductions are first-class methods in HPC – they must be in the language.
  - Data/memory management needs to be more transparent (education?).
  - Move beyond OpenCL/CUDA thinking and support dimensions >3.
- SYCL compiler helps make device lambda usage better.
  - There is a Kokkos compiler effort but it isn't the primary implementation.
- Kokkos@SYCL is a natural next step.
  - SYCL@Kokkos might be an interesting reference implementation…

intel®