Performance Portability in Practice

John Pennycook and Jason Sewall, Intel Corporation

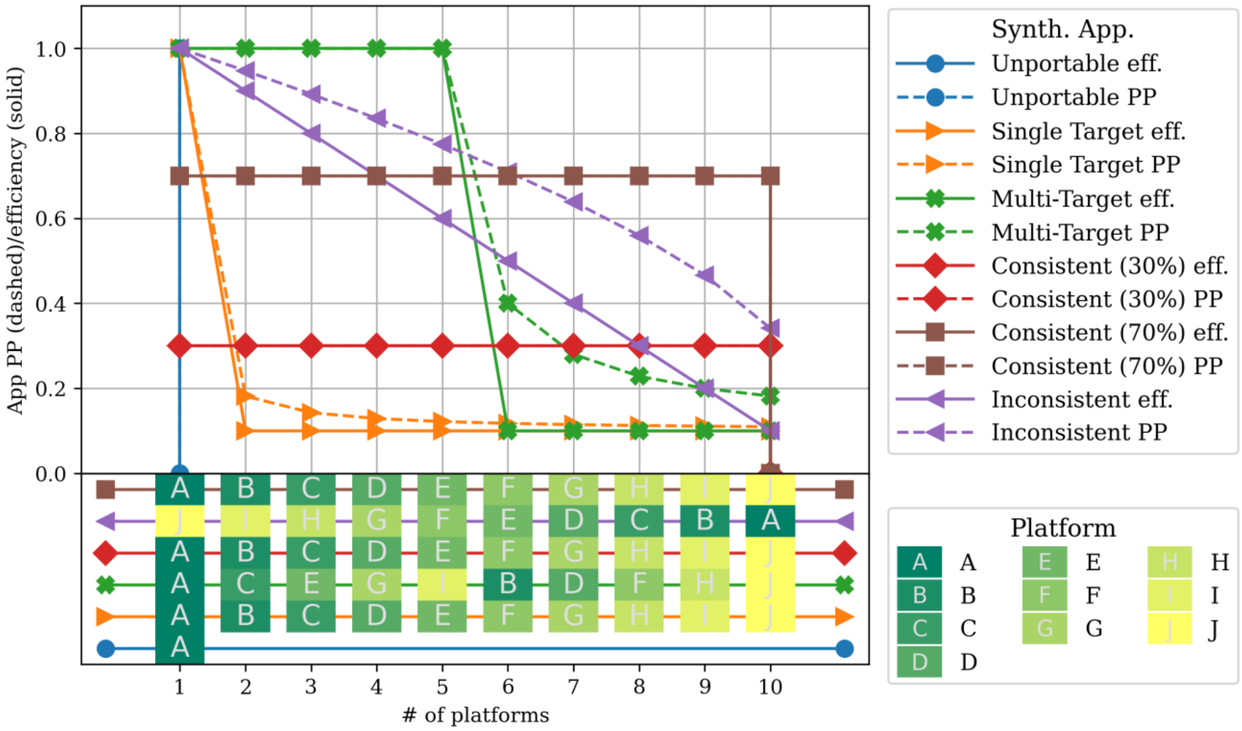
Acknowledgements:
Doug Jacobsen, Google
Tom Deakin and Simon McIntosh-Smith, University of Bristol

What is "Performance Portability"?

"A **measurement** of an application's **performance efficiency** for a given problem that can be executed correctly on all platforms in a given set."

$$\Phi(a,p,H) = \begin{cases} \dfrac{|H|}{\sum_{i\in H}\dfrac{1}{e_i(a,p)}} & \text{if } i \text{ is supported } \forall i \in H \\ 0 & \text{otherwise} \end{cases}$$

- Enables objective comparisons of implementations, algorithms, etc
- See our publications for more detail:
  - "[Implications of a Metric for Performance Portability]"
  - "[Navigating Performance, Portability and Productivity]"



An "efficiency cascade" plot, highlighting performance trends across platforms of interest. Plot your own using these scripts.

What About Productivity?

- Per-platform code increases developer effort:
  - Initial development
  - Debugging
  - Maintenance

- We measure this via “Code Divergence”:

[Codebase 1]

shared.cpp:
```
void whereami()
{
#ifdef CPU
  printf(“CPU\n”);
#else
  printf(“GPU\n”);
#endif
}
```

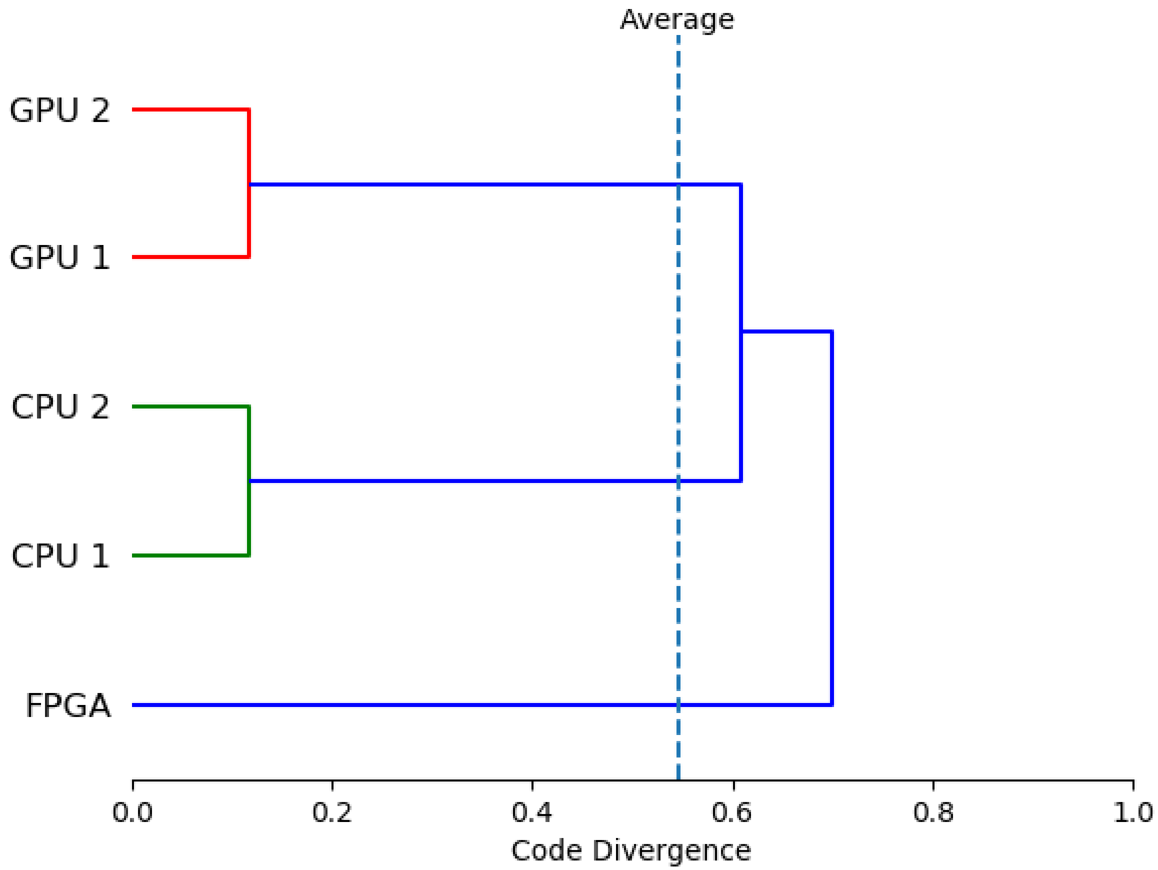
$$\frac{8\text{ lines } - 6\text{ shared lines}}{8\text{ lines}} = 0.25$$

[Codebase 2]

cpu.cpp:
```
void whereami()
{
  printf(“CPU\n”);
}
```

gpu.cpp:
```
void whereami()
{
  printf(“GPU\n”);
}
```

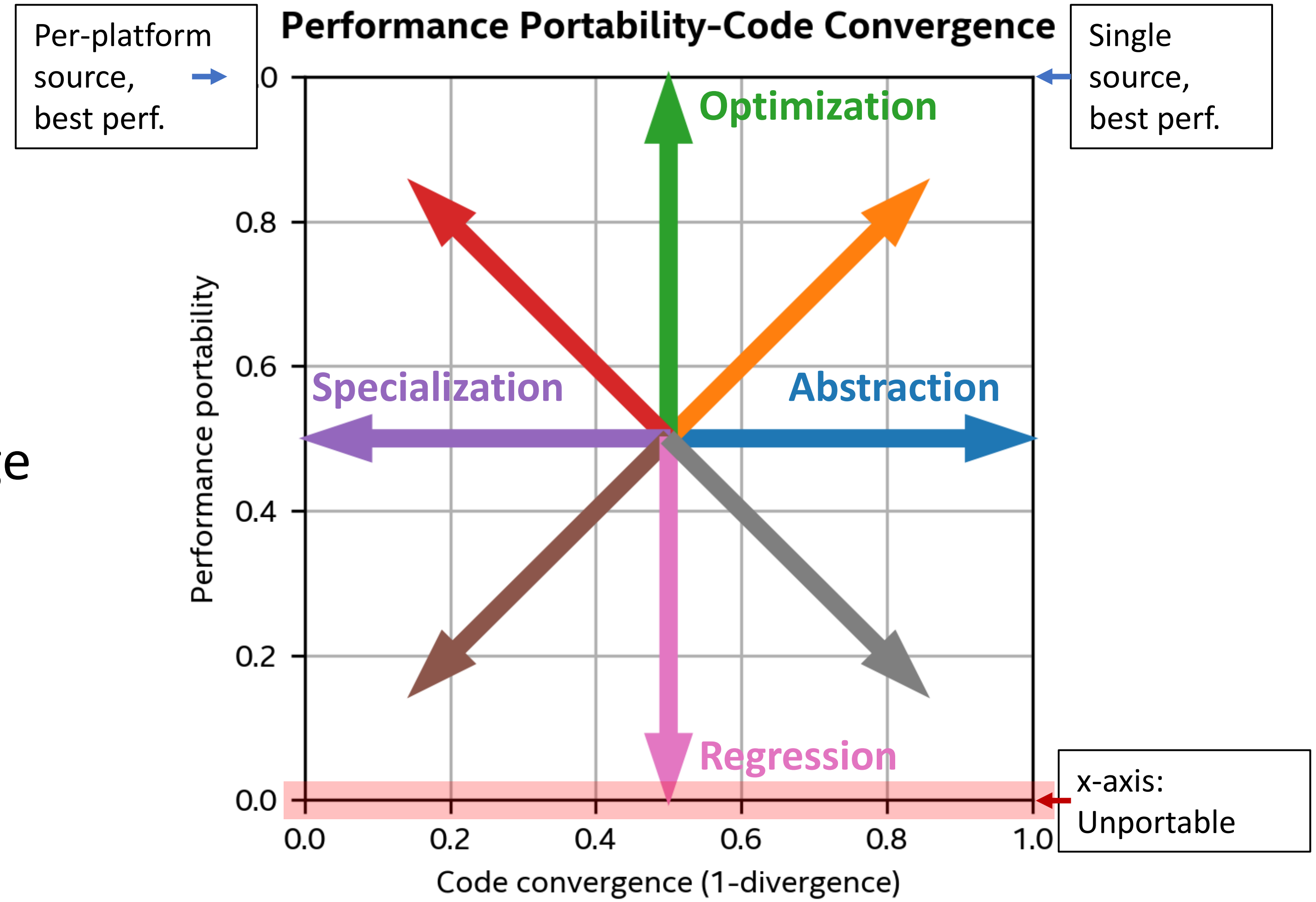
$$\frac{8\text{ lines } - 0\text{ shared lines}}{8\text{ lines}} = 1.0$$



A dendrogram clustering platforms by “code divergence”.
Plot your own using our [Code Base Investigator] tool.

Setting Goals and Tracking Progress

- Trade-offs abound:
  - Abstraction ⇒ Higher Convergence
  - Specialization ⇒ Lower Convergence
  - Optimization ⇒ Higher Performance
- SYCL provides tools to help us manage these trade-offs
- But we must set our own goals (and manage our expectations!)



Device Queries

- Queries allow applications to inspect where they're running and dynamically adapt to the capabilities available.

```
sycl::queue q(sycl::default_selector{});
sycl::device d = q.get_device();
// Process input in the largest chunks we can allocate
uint64_t chunk_size = d.get_info<sycl::info::device::max_mem_alloc_size>();
// Prefer double precision if it's supported
if (d.has(sycl::aspect::fp64)) {
  run<double>(chunk_size);
} else {
  run<float>(chunk_size);
}
```

- Many queries available, e.g.: cache size, work-group and sub-group sizes, availability of scratchpad memory, atomics capabilities

Device Selectors

- Heterogeneous nodes may contain a mix of accelerators; device selectors provide a powerful mechanism to help you quickly select the best.

```
struct MySelector : sycl::device_selector {
  // SYCL 2020 allows this to be a lambda
  int operator()(const sycl::device& d) const {
    int score = 0;
    if (not d.has(sycl::aspect::fp64)) {
      score = -1;
    } // Return negative if required features are not supported
    else if (d.has(sycl::aspect::gpu)) {
      score = 1;
    }  // Return a higher score for GPU devices
    std::cout << "Scoring: " << d.get_info<sycl::info::device::name>() << " " << score << std::endl;
    return score;
  }
};

int main() {
  // Create a queue associated with the selected device
  sycl::queue q(MySelector{});
  std::cout << "Selected: " << q.get_device().get_info<sycl::info::device::name>() << std::endl;
  ...
```

Subdevices (e.g. NUMA Domains)

```
using namespace sycl::info;
constexpr auto by_affinity_domain = partition_property::partition_by_affinity_domain;
auto has_numa_domains = [=](sycl::device &d) {
 auto props = d.get_info<device::partition_properties>();
 auto domains = d.get_info<device::partition_affinity_domains>();
 bool by_affinity = std::find(props.begin(), props.end(), by_affinity_domain) != props.end();
 bool by_numa = std::find(domains.begin(), domains.end(),
                          partition_affinity_domain::numa) != domains.end();
 return by_affinity && by_numa;
};
// Try to split selected device into sub-devices along NUMA boundaries
sycl::device root = sycl::device(sycl::default_selector{});
std::vector<sycl::device> devices;
if (has_numa_domains(root)) {
  devices = root.create_sub_devices<by_affinity_domain>(partition_affinity_domain::numa);
}
else {
  devices = std::vector<sycl::device>{root};
}
std::cout << "Split " << root.get_info<device::name>() << " into " << devices.size()
          << " NUMA domains." << std::endl;
```

Specialization Constants

```
q.submit([&](auto &h) {
  h.template set_specialization_constant<BLOCKSIZE>(blocksize);
  h.parallel_for(nd_range<2>({(size_t)N,(size_t)M},{blocksize,blocksize}),
     [=](nd_item<2> it, kernel_handler kh) {
         const auto BS = kh.template get_specialization_constant<BLOCKSIZE>();
         auto id = it.get_group().get_id();
         const int i = id[1]*BS;
         const int j = id[0]*BS;
         if(BS == 4) {
           do_block_static<4>(it, i, j, M, N, K, C, A, B);
         }
         else if(BS == 8) {
           do_block_static<8>(it, i, j, M, N, K, C, A, B);
         }
         else if(BS == 16) {
           do_block_static<16>(it, i, j, M, N, K, C, A, B);
         }
         else {
           do_block_dyn(BS, it, i, j, M, N, K, C, A, B);
         }
      });
  }).wait();
```

- JIT compilation-enabled optimization tool
- Set dynamically in handler
- Act as constant during JIT lowering

Group Algorithms

- Syntax and semantics based on ISO C++ algorithms
  - `any_of, all_of, none_of, shift_left, shift_right, reduce, exclusive_scan, inclusive_scan`
  - SYCL specific: `permute`, `select`

```
// Apply a group algorithm to any number of values, described by an iterator range
// The work-group reduces all inputValues and each work-item works on part of the range
int* first = inputValues.get_pointer();
int* last = first + 1024;
int sum = sycl::joint_reduce(it.get_group(), first, last, plus<>());

// Apply a group algorithm to a set of values held directly by work-items
// The work-group reduces a number of values equal to the size of the group
// and each work-item provides one value
int partial_sum = sycl::reduce_over_group(it.get_group(),
                                          inputValues[it.get_linear_id()], plus<>());
```

DPC++ Extension: invoke_simd

- Exploring ways to provide interoperability between SPMD and SIMD programming

```
int popcount(sycl::ext::oneapi::experimental::simd_mask<bool, 8> mask) {
  // Utilize SIMD APIs with no SPMD equivalent, e.g., popcount
  return sycl::ext::oneapi::experimental::popcount(mask);
}
...
q.parallel_for(sycl::nd_range<1>{N},
               sycl::nd_item<1> it) [[sycl::reqd_sub_group_size(8)]] {
  // Express algorithm in SPMD where convenient
  bool cond = (it.get_local_id() % 2);
  // Pass control to explicitly vectorized SIMD function
  sycl::sub_group sg = it.get_sub_group();
  int ct = sycl::ext::oneapi::experimental::invoke_simd(sg, popcount, cond);
  // Resume SPMD coding
  ...
}
```


Draft extension at https://github.com/intel/llvm/blob/sycl/sycl/doc/extensions/InvokeSIMD/InvokeSIMD.asciidoc

Summary and Call to Action

- SYCL exposes many features to help us achieve our desired balance of performance, portability and productivity

- Achieving high levels of performance (portability) may require non-trivial effort; what is “acceptable” effort is subjective

- Help us improve SYCL and implementations:
  - SYCL Specification: https://github.com/KhronosGroup/SYCL-Docs
  - Intel DPC++: https://github.com/intel/llvm
  - *TODO: LINKS FROM CODEPLAY, XILINX, AKSEL HERE?*