Performance Portability of Kernel-based Abstractions

John Pennycook, Intel HPC Ecosystem and Applications Team

DOE COE Performance Portability Meeting, April 2016, Arizona

Acknowledgements
Alex Duran, Jeongnim Kim, Amrita Mathuriya, Larry Meadows, Roland Schulz, Jason Sewall, Dayle Smith

Performance Portability

- “Performance portability” isn’t well defined.
  - What level of performance is satisfactory?
  - How much “compiler magic” is expected?

- The presenter’s definition is something like:

  “An **application** is ‘performance portable’ if it achieves a consistent level of performance across platforms (**relative to the best known implementation on each platform**).”

The Problem with Performance Portability

- When writing a new HPC application, we're faced with three decisions:

Decision	**Example Choices**	**Impact**
Programming Model	Assembly, Intrinsics, OpenMP*	Portability
Algorithm	Quicksort, Mergesort, Bitonic	Performance
Implementation	Cache blocking, compiler opts	Performance

- "Which model provides performance portability?" is the **wrong** question.
  - The answer does not directly impact performance!
    (beyond runtime overheads or incompatibility with certain algorithms/implementations)

- Unfortunately, this **does** mean that we should expect to maintain different algorithms and implementations for each architecture†.

† In the presenter's opinion.

“Parallel Kernels” as an Abstraction

- “Parallel kernel” programming is a **model** that can expose parallelism well:

```
for all elements in domain:
  do something (potentially in parallel)
for all elements in domain:
  do something else (potentially in parallel)
```

- One parallel language construct per kernel is only one **implementation**.
  - We need to separate the language from architectural considerations.
  - We should not be tempted to create tools with “**an** OpenMP backend” .

Architectural Considerations - Synchronization

Offload to Accelerator

- Accelerator => Fixed functionality
- Synchronization = Host-device handshake

- Compatible with all devices†
- Synchronization required when constructs are not supported by the accelerator

```
for (int t = 0; t < tfinal; ++t)
{
  #pragma omp target teams distribute parallel for
  for (int i = 0; i < N; ++i)
  {
    ...
  }
}
```

```
#pragma omp target
for (int t = 0; t < tfinal; ++t)
{
  #pragma omp parallel for
  for (int i = 0; i < N; ++i)
  {
    ...
  }
}
```

Offload to Another Node

- Another Node => Same/similar functionality to the host
- Synchronization = Network communication?

- Not compatible with all devices
- Synchronization required only when the algorithm requires host-device communication

† In theory.

Architectural Considerations - SIMD

```
#pragma omp parallel for simd
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < M[i]; ++j)
    {
        ...
    }
}
```

Combined Threading + SIMD

- Need: N >> # threads * SIMD width
- Best data layout is likely:
  - i-contiguous for SIMD
  - j-contiguous for cache

- SIMD efficiency impacted by different M values per i

Separate Threading + SIMD

- Need: N > # threads; M > # SIMD width
- Best data layout is likely j-contiguous

- SIMD efficiency not impacted by different M
- Supports certain dependencies in j loop

```
#pragma omp parallel for
for (int i = 0; i < N; ++i)
{
    #pragma omp simd simdlen(4)
    for (int j = 0; j < M[i]; ++j)
    {
        ...
    }
}
```

Architectural Considerations - Threading

```
#pragma omp parallel for collapse(2)
for (int i = 0; i < N; ++i)
{
    for (int j = 0; j < N; ++j)
    {
        ...
    }
}
```

OpenMP Parallel Loops

- Domain implicitly flattened to N*N and then decomposed
- All threads operate on part of the same, shared, array

- Exposes all N*N iterations to thread-level parallelism
- Scheduling is defined by the compiler/runtime

Tiling/Domain Decomposition

- Domain explicitly divided by the programmer
- Supports both:
    1. Tiles as part of a shared array; and
    2. Tiles as individual arrays

- Scheduling is defined by the user
- May incur additional memory overhead

```
#pragma omp parallel for
for (int t = 0; t < ntiles; ++t)
{
    Tile tile = tiles[t];
    for (int i = tile.i0; i < tile.iN; ++i)
    {
        for (int j = tile.j0; j < tile.jN; ++j)
        {
            ...
        }
    }
}
```

Architectural Considerations - Bandwidth

- Naïve kernels are likely to be bandwidth-bound and bandwidth-inefficient (**everywhere**):

Read = O(4N) + Write = O(2N)

```
#pragma omp parallel for simd
for (int i = 0; i < N; ++i)
{
  v[i] += dtforce * f[i];
}
#pragma omp parallel for simd
for (int i = 0; i < N; ++i)
{
  x[i] += dt * v[i];
}
```

Read = O(3N) + Write = O(2N)

```
#pragma omp parallel for simd
for (int i = 0; i < N; ++i)
{
  v[i] += dtforce * f[i];
  x[i] += dt * v[i];
}
```

- Cache-blocking, scratchpad arrays, etc can help you to optimize the memory access pattern of an individual kernel, but they can't fix this.
- In the general case, cross-kernel optimizations requires in-depth knowledge of the code (which compilers and analysis tools don't have).

“Parallel Kernels” + Elemental Functions

- Extending the abstraction (with **elemental functions**) simplifies maintenance:

```
for (int i = 0; i < natoms; ++i)
{
  v[i] = update_velocity(v[i], f[i]);
}

for (int i = 0; i < natoms; ++i)
{
  x[i] = update_position(x[i], v[i]);
}
```

```
for (int i = 0; i < natoms; ++i)
{
  v[i] = update_velocity(v[i], f[i]);
  x[i] = update_position(x[i], v[i]);
}
```

```
#pragma omp task depend(out: v[i])
v[i] = update_velocity(v[i], f[i]);
#pragma omp task depend(in: v[i]) depend(out: x[i])
x[i] = update_position(x[i], v[i]);
```

- We have to write new implementations for each architecture of interest, and we’ll likely have to do so again for a new architecture in the future.
  – But the science is the same, and our execution “patterns” can be maintained once.

Inspired by “Æcute” -- see Howes, Lee W. et al. "Deriving Efficient Data Movement from Decoupled Access/Execute Specifications." *High Performance Embedded Architectures and Compilers*. Springer Berlin Heidelberg, 2009. 168-182.

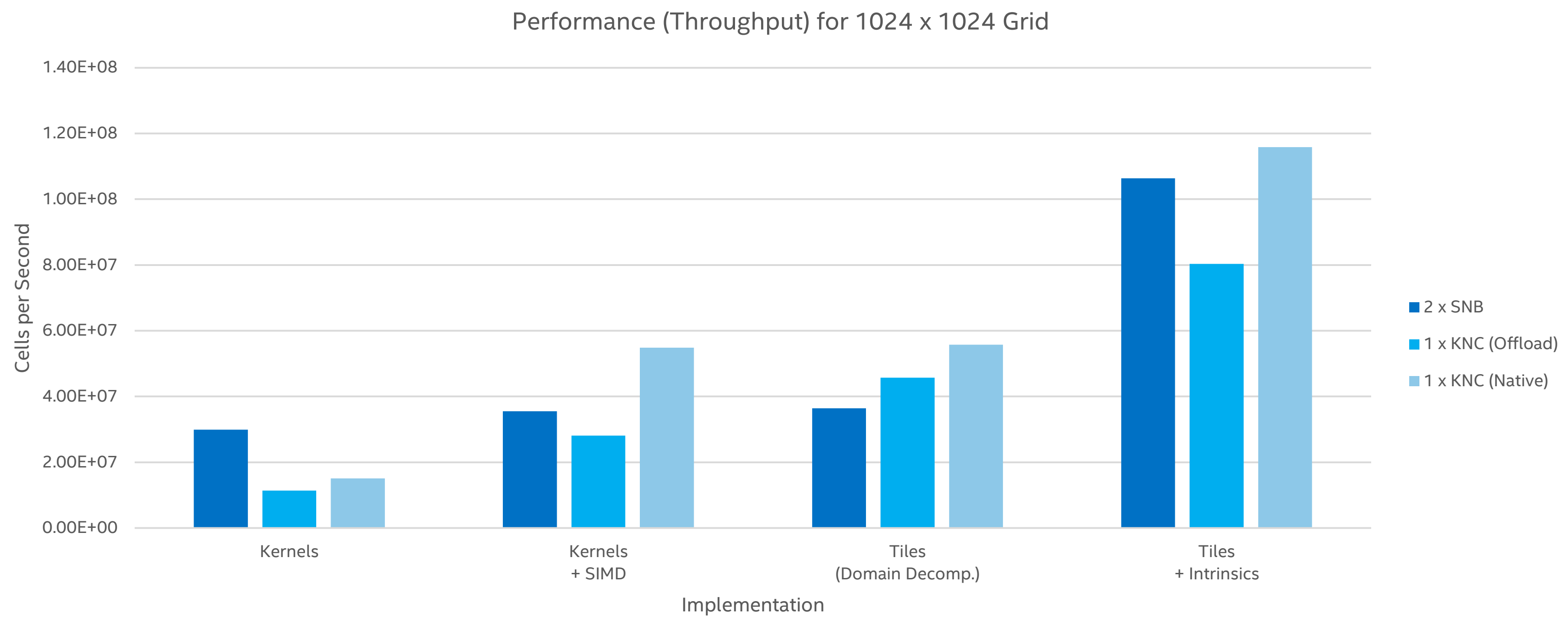
Example – CEA's Hydro2D

- Dimensionally split shock-hydro code using Godunov's scheme.
  - https://github.com/HydroBench/Hydro/

- Kernels operating on a mix of cells and interfaces.
  - Not a one-to-one mapping from kernels to elemental functions

- Fairly typical synchronization pattern:
  - Compute interface properties from cells
  - Synchronize
  - Compute cell properties from interfaces

Case Study: CEA's Hydro2D



Summary

- Achieving “performance portability” relies on a good abstraction
  - We shouldn’t blame languages or hardware, but our own abstractions
- Parallel kernels **can** be a useful abstraction
  - But only if we pay careful attention to how they are scheduled
- Elemental functions provide another level of abstraction (without a DSL!)
  - “write-most-parts-once”

Legal Notices and Disclaimers

intel®