Codesign In Action

- Center for Exascale Simulation of Advanced Reactors (CESAR)
- Contributions from specialists in entire HPC stack
- Birdectional involvement
  - For example, programming models inform design of apps AND vice versa.
Many-core Programming Models

- For programming models...
  - Can diverse architectures be programmed with a common set of abstractions?

- For our apps...
  - What changes are needed to accommodate the different programming models?
  - What are the performance and portability tradeoffs?
Different backends are abstracted into common kernel languages.

Front-end language calls kernel through OCCA API.

Kernels can be written in a number of languages.

OCCA parser creates IR and invokes backend compiler.

We’ll see examples later!
Nuclear Reactor Simulation

...especially neutron transport
Some Aspects of Reactor Simulation

**Neutron Transport**
- Fission in fuel, including heat generation.
- Also used for medical imaging, radiation shielding, etc.

**Fluid Dynamics**
- Distribution of heat in liquid coolant.
- Used for LOTS of other applications.
Neutron Cross-sections

Neutron cross sections are physical quantities that describe the interaction of neutrons with matter...

- **Microscopic cross-sections** ("micro xs") describe the interaction of an incident neutron with a target nuclide.
  - ~ the probability that a neutron interact with the given nuclide
  - Depend on neutron energy and nuclide identity

- **Macroscopic cross-sections** ("macro xs") describe the interactions of a neutron as it travels through a material composed of many nuclides.
  - ~ the mean free path of of a neutron through the material
  - Depend on neutron energy and material composition
  - Density-weighted average of component micro xs’s

Deskribed for different interactions (absorption, elastic scattering, fission, etc)
Cross-section Representations

- **Micro XS**
  - **Continuous energy**: pointwise defined on a dense, non-uniform energy grid.
  - **Energy groups**: energy space is more coarsely discretized.

- **Macro XS**
  - For continuous-energy, must be **computed at runtime**.
  - For energy groups, can be **precomputed**.
Many methods for neutron transport, including...

- **Monte Carlo (MC)**
  - Obtain estimators from many independent neutron histories.
  - Energy space is continuously-valued.
  - Closer to realistic, hi-fi simulations.
  - Memory bound and convergence requires a large number of histories.

- **Method of Characteristics (MOC)**
  - Attenuate neutron fluxes across independent spatial tracks.
  - Energy space is discretized.
  - 3D MOC is still under development.
  - Higher computational intensity and is easier to vectorize.

- Lots of inherent parallelism in both methods!
Monte Carlo
Monte Carlo Algorithm and Apps

for each neutron in batch
repeat
  for each nuclide in local material
    for each interaction type
      lookup discrete-valued micro xs
      interpolate continuously-valued micro xs
      accumulate micro xs into macro xs
      update neutron’s position and energy
  until neutron is absorbed or escapes boundaries

- **Micro XS lookups** consume up to 80% of total runtime!
  - Particle state is unpredictable.
  - Lookups have poor locality.
  - Latency-bound for few cores, approaches bandwidth limit for many cores.

- **Mini-app (XSBench)** can:
  - abstract away physics
  - simulate xs lookups
  - faithfully replicate performance of full app
#pragma omp parallel private(...), shared(...)  
#pragma acc data copyin(...), copy(...)  
{

#pragma omp for schedule(dynamic)  
#pragma acc kernels loop gang, vector  
for(int i = 0; i < n_neutrons; i++)  
{
...

#pragma acc loop seq  
for (int nuc_id=0; nuc_id < num_nucs(material); nuc_id++)  
{
  ...
  get bounding gridpoints for this nuclide’s micro_xs (hi, lo) ...  
  // Get interpolation factor  
  f = (hi->energy - p_energy) / (hi->energy - lo->energy)  
  // For each interaction, interpolate continuous micro-xs  
  macro_xs[0] += conc * (hi->micro_xs[0] - f * (hi->micro_xs[0] - lo->micro_xs[0]));  
  macro_xs[2] += ...  
  macro_xs[3] += ...  
  macro_xs[4] += ...

  // Get interpolation factor
  f = ...
  // For each interaction, interpolate continuous macro-xs
  macro_xs[0] += ...
  macro_xs[1] += ...
  macro_xs[2] += ...
  macro_xs[3] += ...
  macro_xs[4] += ...

}

}

□ Coarse-grained parallelism over neutron loop.

□ In OpenMP, parallelizing only the outer loop performs better than nested parallelism.

□ In OpenACC, the performance is also better when the inner loops are sequential.
CUDA

```
__global__ void lookup_kernel(...)
{
  int global_id = blockIdx.x * blockDim.x + threadIdx.x
  if (global_id < n_neutrons)
  {
    for (int nuc_id=0; nuc_id < num_nucs(material); nuc_id++)
    {
      ... get bounding gridpoints for this nuclide’s micro-xs (hi, lo) ...
      __ldg(lo->energy); __ldg(lo->micro_xs[1]); __ldg(lo->micro_xs[2]); ...
      __ldg(hi->energy); __ldg(hi->micro_xs[1]); __ldg(hi->micro_xs[2]); ...
      // Get interpolation factor
      double f = (hi->energy - p_energy) / (hi->energy - lo->energy)
      // For each interaction, interpolate continuous micro-xs
      macro_xs[0] +=
        conc * (hi->micro_xs[0] - f * (hi->micro_xs[0] - lo->micro_xs[0]));
      macro_xs[1] += ...
      ...
    }
  }
}
```

- Also coarse-grained.
- Each thread performs a different lookup.
- How do we exploit capabilities of GPU?
  - Threads stall on data dependencies (not memory access).
  - Queue up many memory accesses before the data are needed.
OCCA

- Similar to CUDA, but parallelism is more explicit.
- When compiled for OpenMP, outer and inner loops are coalesced.
- When compiled for CUDA, outer is mapped to blocks and inner loop is mapped to threads.
- OCCA translates to coarse-grained parallelism for both backends.

```c
for (outer_id=0; outer_id < outer_dim; outer_id++; outer0)
{
  for (inner_id=0; inner_id < inner_dim; inner_id++; inner0)
  {
    int global_id = outer_id * outer_dim + inner_id;
    if (global_id < n_neutrons)
    {
      ...
      for (int nuc_id=0; nuc_id < num_nucs(material); nuc_id++)
      {
        ...
        get bounding gridpoints for this nuclide’s micro_xs (hi, lo) ...
        directLoad(lo->energy); directLoad(lo->micro_xs[1]); ...
        directLoad(hi->energy); directLoad(hi->micro_xs[1]); ...
        // Get interpolation factor
        double f = (hi->energy - p_energy) / (hi->energy - lo->energy)
        // For each interaction, interpolate continuous micro-xs
        macro_xs[0] +=
          conc * (hi->micro_xs[0] - f * (hi->micro_xs[0] - lo->micro_xs[0]));
        macro_xs[1] += ...;
      }
    }
  }
}
```
XSbench Performance

- Xeon Sandy Bridge (2X E5-2650, 32 threads)
- Xeon Haswell (2X E5-2699, 72 threads)
- Tesla K40m

Time per lookup (ns)
Method of Characteristics

...different portability and performance...
Geometric Decomposition in 3D MOC

Stacked 2D Planes

2D Tracks

Segments

Attenuate Flux over Energy Groups
MOC Algorithm and Apps

- The larger app (SimpleMOC) simulates 3D geometry, including domain decomposition.

- The proxy-app (SimpleMOC-kernel) randomly selects segments and proceeds similarly.

- Attenuating flux across energy groups offers opportunities for SIMD.
Fine-grained parallelism.

Outer loop (segments) is parallelized with OpenMP.

Inner loop (energy groups) is fissioned and vectorized.

- ~50 FLOPS to attenuate flux, fissioned into 12 SIMD loops.
- Intel compiler can vectorize all 12 loops.
- GNU compiler can vectorize 3 loops.
CUDA

```c
__global__ void kernel(...) {
    int blockId = blockIdx.y * blockDim.x + blockIdx.x;
    if (blockId >= n_segments) return;

    // It implied that each thread is an energy group
    // int egroup = threadIdx.x

    // SIMT over energy groups is implied
    ...
    float tau = sigT * ds;
    float sigT2 = sigT * sigT;
    ...
    float flux_integral = (q0 * tau + (sigT * ...)
    ...
}
```

- Similar parallelism, expressed differently...
  - Outer loop (segments) is mapped to blocks
  - Inner loop (energy groups) are mapped to threads.
- Within a warp, energy groups are attenuated in SIMT.
  - Same goal as CPU.
occaKernel void kernel(...) {
  for (int outerId1 = 0; outerId1 < outerDim1; outerId1++) { 
    for (int outerId0 = 0; outerId0 < outerDim0; outerId0++) {
      int outerId = outerId1 * outerDim0 + outerId0;
      if (outerId >= n_segments) return;
      
      for (int innerId0 = 0; innerId0 < innerDim0; innerId0++) {
        int g = innerId0;
        ...
        float tau = sigT * ds;
        float sigT2 = sigT * sigT;
        ...
        float flux_integral = (q0 * tau + (sigT * ...)
      }
    }
  }
}

- Outer loop over segments, inner loop over energy groups.
- When compiled for CUDA, energy groups are computed in SIMT.
- When compiled for OpenMP, the energy groups are NOT vectorized.
- Compiler cannot resolve dependencies without loop fission.
- Question: how do we write an efficient and portable kernel?
SimpleMOC-kernel Performance

- Xeon Sandy Bridge (2X E5-2650, 32 threads)
- Xeon Haswell (2X E5-2699, 72 threads)
- Tesla K40m

Time per lookup (ns)

- OpenMP (Intel)
- OpenMP (GNU)
- OCCA (GNU)
- CUDA
- OpenACC
SimpleMOC-kernel CPU Vectorization

Xeon Sandy Bridge
(2X E5-2650, 32 threads)

Xeon Haswell
(2X E5-2699, 72 threads)

Time per lookup (ns)
Conclusions

- Monte Carlo App:
  - Coarse-grained parallelism
  - Easy to express portably

- Method of Characteristics App:
  - Fine-grained parallelism
  - Harder to express with portable performance
  - We live in an interesting time and can expect fascinating solutions!


Repositories

- XSbench, https://github.com/ANL-CESAR/XSBench
- OCCA2, https://github.com/tcew/OCCA2