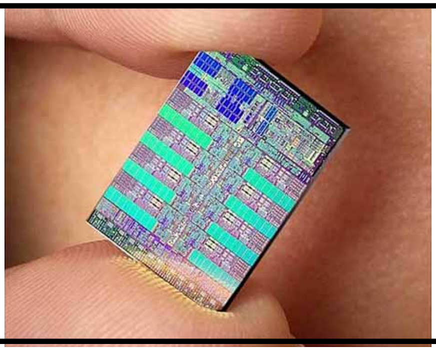
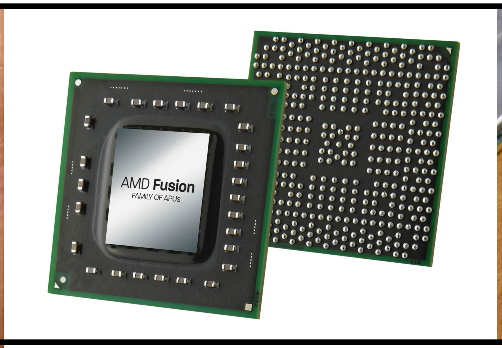
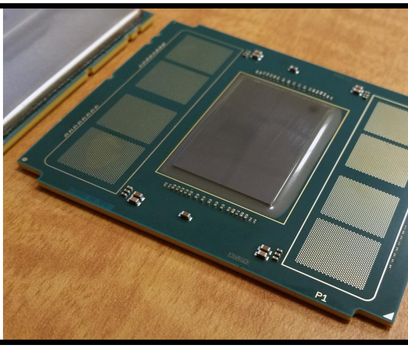
Exceptional service in the national interest

Sandia National Laboratories







Profiling Kokkos Applications

Christian Trott

crtrott@sandia.gov

Center for Computing Research

Sandia National Laboratories, NM

SAND2017-4686 PE

U.S. DEPARTMENT OF ENERGY

NNSA National Nuclear Security Administration

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology and Engineering Solutions of Sandia, LLC., a wholly owned subsidiary of Honeywell International, Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

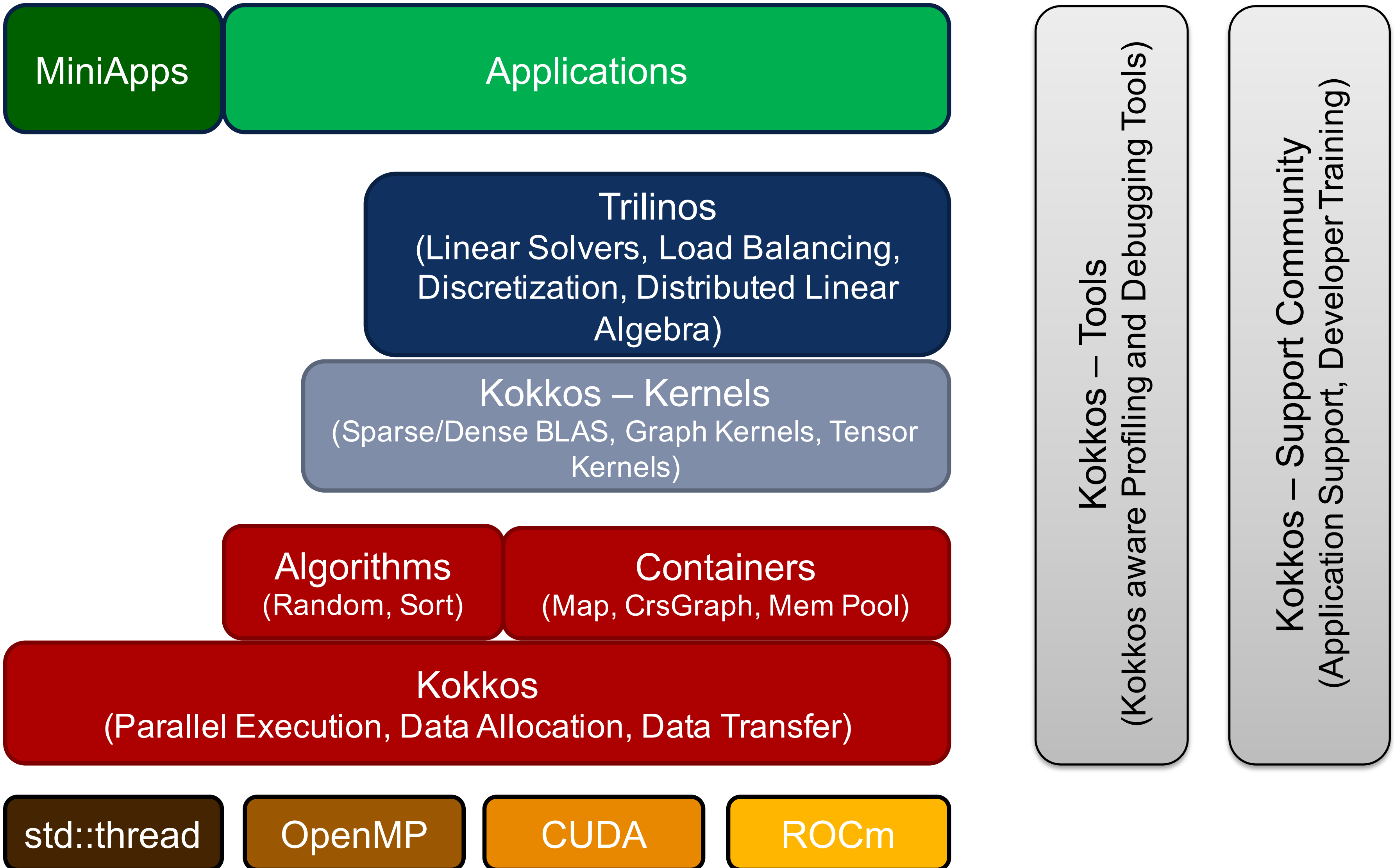
C++ The Bane of Profiling Tools

- It is hard to understand C++ code for a compiler
  - Template Metaprogramming
  - Function Pointers
  - Inheritance
  - Arbitrary aliasing
- It is even harder for a Performance Analysis Tool
  - Most come from a Fortran history
- Abstraction Models make it all worse
  - E.g. only one place where the actual OpenMP loop is
  - Really complex type names which may exceed internal typename length limits
- And we want this across all Platforms …

Abstractions to the Win ?!

- Abstractions can also help us: Instrumentation
- KokkosTools provide built-in instrumentation for Kokkos applications
  - By default enabled on most platforms
- This Instrumentation knows about Kokkos abstractions
  - Get information organized by Kokkos constructs (Parallel Regions, Allocations in Memory Spaces, etc.)
- Enables Meta Instrumentation for Third Party Tools
  - Provide information to Vtune, Nsight, …
- Easy to use Tools Provide Basic Information accross all Platforms
  - Kernel and Region Times, Memory Utilization, Allocation and Deallocation Frequency, ...

Building an EcoSystem



KokkosTools: github.com/kokkos/kokkos-tools

- Utilities
  - **KernelFilter:** Enable/Disable Profiling for a selection of Kernels
- Kernel Inspection
  - **KernelLogger:** Runtime information about entering/leaving Kernels and Regions
  - **KernelTimer:** Postprocessing information about Kernel and Region Times
- Memory Analysis
  - **MemoryHighWater:** Maximum Memory Footprint over whole run
  - **MemoryUsage:** Per Memory Space Utilization Timeline
  - **MemoryEvents:** Per Memory Space Allocation and Deallocations
- Third Party Connector
  - **VTune Connector:** Mark Kernels as Frames inside of Vtune
  - **VTune Focused Connector:** Mark Kernels as Frames + start/stop profiling

How to Use the Tools

- Checkout from github.com/kokkos/kokkos-tools
- Documentation in a Wiki
- Go to src/tools/TOOLNAME and build the tool
  - On most platforms typing ”make” is enough
- Before running your code set environment variable
  - export KOKKOS_PROFILE_LIBRARY=/PATH/TO/TOOLS/LIBRARY
- Analyze output
  - Some tools print to screen, some write per-process files
  - Some tools have readers for binary output files
- How does it work internally?
  - Instrumentation is always active, but internal function pointers are NULL
  - At Kokkos::initialize tool library is dynamically loaded, and function pointers are set

Typical Approach

- Run KernelTimer
  - Check if majority of time is in Kernels
  - Check for HotSpot Kernels
- Run MemoryUsage
  - Check where your memory utilization is coming from
  - Check total number of entries to see if frequent alloc/dealloc could be an issue
- If frequent allocations are an issue: run MemoryEvents
  - Figure out which allocations are causing the issue
  - Less than 1000 per second per socket is usually no issue
- To find unaccounted time: put region markers into code
  - Compare region times with kernel times
- Use Connector tools to help investigate individual kernels

The Tools

- See Wiki

Exercise

- Use MiniMD a Molecular Dynamics ProxyApp
  - git clone git@github.com:crtrott/miniMD
  - git checkout profiling-exercise
- Basic Molecular Dynamics Importance of Kernels:
  - Force Calculation
  - NeighborList Construction
  - Communication
  - Other stuff: (Integration, Particle Sorting etc.)
- This variant has a problem hidden
  - Use Kokkos Tools to find the issue
  - Follow the typical approach lined out before
  - The main time integration loop is in integrate.cpp::run()
  - Compile:
    - 'make'
    - for CUDA export OMPI_CXX=${KOKKOS}/bin/nvcc_wrapper
    - Build with fake MPI: got to MPI-Stubs type 'make', build miniMD with HAVE_MPI=no

(NVIDIA) GPU Profiling

- Dominant Performance Bottlenecks:
  - Occupancy
  - Memory Bandwidth
  - Memory Efficiency
  - Memory Load/Store Slots
  - Availability of Instruction Parallelism
- Visual Profiler (nvvp or as part of nsight)
  - Guided Analysis
- Use nvprof to collect data on commandline
  - Generally same information as in the Visual Profiler
  - nvprof [OPTIONS] ./Executable [OPTIONS]
    - --print-gpu-summary : Summary of Kernels, and data transfers
    - --print-gpu-trace: timeline of kernels and data transfers
    - --query-metrics: list of collectable events
    - -m [EVENTS]: set events to be collected
    - --kernels [KERNELS]: restrict profiling to specified kernels

Important Metrics – Occupancy/Mem

- **achieved_occupancy**:
  - Actually reached occupancy
  - Cause1: high register pressure (check with --print-gpu-trace)
  - Cause2: high shared memory usage (check with --print-gpu-trace)
  - Cause3: low total available parallelism (too few blocks)
- ****_throughput**: Bandwidth for different parts of the memory subsystem
  - **dram_[read/write]**: device memory traffic including ECC
  - **[gld/gst]**: global memory access, could be cached (this is larger than requested due to efficiency)
  - **[gld/gst]**_requested: the memory throughput of things the code actually wants
  - **l2_l1_read, l2_tex_[read/write],l2_atomic**: L2 Cache Throughput by Source
  - **local_[load/store]**: data traffic due to register spilling
  - **shared_[load/store]**: shared memory (team scratch level 0)
- ****_efficiency**: different efficiency metrics
  - **[gld/gst]:** global memory access (coalesced access = 100%)
  - **shared:** shared memory loads
- ****_hit_rate:** Cache hit rates
  - **l1_cache_[global/local]:** Hit rate in L1 Cache due to global/local load store
  - **tex_cache:** Hit rate for texture fetches
  - **l2_l1_[read]:** Hit rate for all L1 misses in L2
  - **l2_tex_read:** Hit rate for texture misses in L2

Important Metrics – Compute

- ****_efficiency**:
  - **sm**: multiprocessors are active (small means not enough work launched in a kernel)
  - **warp_execution**: active threads vs non-active threads due to branching (small means divergence)
  - **branch**: non-divergent vs total branches (small means divergence in kernel)
  - **flop_[sp/dp]:** achieved single/double precision peak flop/s fraction
- **stall_****: Reasons why a warp does not execute an instruction
  - **inst_fetch:** instructions are not yet loaded,
    - very unlikely to be an issue, if it is think about breaking up kernel into smaller ones
  - **memory_dependency:** waiting for a load
    - typical sign for memory bandwidth limitation
    - Use less memory, spread out loads more if possible to overlap with compute
  - **exec_dependency:** can't execute because prior instruction not done
    - find and expose instruction parallelism
  - **memory_throttle:** no load/store slots available
    - If this happen without memory_dependency data access patterns are usually bad
  - **sync:** warps waiting for other warps at a barrier
    - Check if barriers are necessary
  - **pipe_busy:** compute operation pipe is busy
    - Rarely an issue, except when making haevy use of special function units
  - **not_selected:** The "good" stall, more ready threads are availabe than slots are to be filled
    - If you see this as the primary stall reason you have a code which is either artificial or a Gordon Bell candidate

A Case Study I

- A Parametrized Benchmark Code
  - Very fancy vector addition with additional math
- Parameters:
  - N: control total work (RT)
  - M: control data reusage origin (RT)
  - S: control data stride (RT)
  - R: control data reusage (RT)
  - K: control instruction parallelism (CT)
  - F: control Flops/Bytes ratio (RT)

```
View<double***> a("A",N,M,S), b("B",N,M,S),
                c("C",N,M,S);
// Loop over blocks
for(int i=0; i<N; i++) {
  // Repeat work on block to control cache reusage
  for(int r=0; r<R; r++) {
    // Loop over block
    for(int j=0; j<M; j++) {
      a_1 = a(i,j,0);
      b_1 = b(i,j,0);
      // Repeat for instruction parallelism
      a_K = a(i,j,0);
      b_K = a(i,j,0);
      // Loop to add more flops
      for(int f=0; f<F; f++) {
        a_1 += b_1;
        // Repeat for instruction parallelism
        a_K += b_K;
      }
      c(i,j,0) = a_1 + /*...*/ a_K;
    }
  }
}
```

A Case Study II

- Exercise I
  - Find settings to measure hardware global and cache bandwidth
- Exercise II
  - Find setting to maximize flop rate
- Exercise III
  - Find settings to make each of the stall reasons the primary stall reasons

Sandia National Laboratories

Exceptional service in the national interest

http://www.github.com/kokkos