DS 7347
High-Performance Computing (HPC) and Data Science
Session 26

Robert Kalescky
Adjunct Professor of Data Science
HPC Research Scientist

July 26, 2022

Research and Data Sciences Services
Office of Information Technology
Center for Research Computing
Southern Methodist University

Outline

Session Questions

Session Questions

Tuesday

Generally, what's the relationship between memory speed and size? Describe the hierarchy.

Thursday

How would you parallelize painting a room and building a car? What is the maximum amount of work that can be done in parallel?

Parallelization Limits

Concurrency and Parallelization

Concurrency

- Structure to handle multiple tasks
- Tasks can run serially or in parallel

Parallelization

- Multiple tasks running at the same time
- Parallelization is a special case of concurrency

Laws Concerning Parallelization

Moore's Law Performance doubling every 18 months

Amdahl's Law There's a upper limit to adding more resources

Gustafson's Law There's a upper limit to adding more resources

Brooks's Law There's a upper limit to adding more human resources

Law of Diminishing Returns There's an ideal set of resources for a problem

C++ Parallel Algorithms

- C++17 introductions high-level parallel versions of many algorithms found in the Standard Template Library
- Several new algorithms were included specifically to aid in using the new parallel algorithms, *e.g.* `std::reduce` and `std::transform_reduce`
- Each parallel algorithm has an execution policy that defines how the algorithm is parallelized

Execution Policies

The C++17 standard defines three execution policies:

`std::execution::seq` Sequential execution, *i.e.* no parallelism

`std::execution::par` Parallel execution via one or more threads

`std::execution::par_unseq` Parallel execution via one or more threads with each thread possibly vectorized

Parallel Execution

When using an execution policy other than `std::execution::seq`:

- The compiler **may** execute the algorithm in parallel when possible and advantageous (some conforming C++17 inplementations may ignore the parallelization flag and run via `std::execution::seq`)
- It is up to **you** to make sure that the algorithm and data are safe to be run in parallel

Figure 1: Components of the oneAPI 2021 base installation.

oneAPI DPC++/C++ Compiler

- Standards-based C++ compiler based on the open source LLVM compiler infrastructure
    - Full support up through C++17
    - Initial support for C++20
- SYCL compiler
- DPC++ compiler (SYCL with Intel extensions)
- Partial support for OpenMP 4.5 and 5.0 offloading

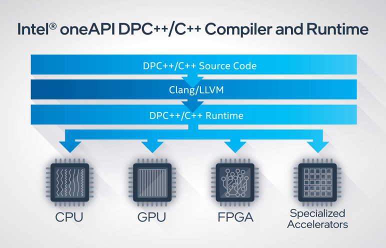

Figure 2: oneAPI DPC++ and C++ compiler and runtime.

oneAPI DPC++ Library

- Productivity APIs for heterogeneous computing
- Optimized standards-based and familiar APIs:
  - C++ STL
  - Parallel STL (PSTL)
  - Boost.Compute
  - Standard SYCL
- Custom iterators for parallel algorithms
- Use device and host containers to target GPUs and FPGAs or run your code across multi-node CPUs

oneAPI Threading Building Blocks

- Simplifies the work of adding parallelism to complex applications
- Runtime library automatically maps logical parallelism onto threads, making more efficient use of node resources
- Scalable data-parallel programming
  - Multiple threads to work on different parts of a collection
  - Scales well to larger numbers of processors by dividing the collection into smaller pieces
  - Program performance increases as processors and accelerators are added

Intel Advisor

- Design and analysis tool for achieving high application performance
- Helps identify efficient threading, vectorization, and memory use, and GPU offload
- Supports C, C++, Fortran, DPC++, OpenMP, and Python.
- Intel Advisor features:
  - Offload Advisor
  - Automated Roofline Analysis
  - Vectorization Advisor
  - Threading Advisor
  - Flow Graph Analyzer

Intel VTune Profiler

- Application performance profiler for CPU, GPU, and FPGA
- Supports DPC++, C, C++, Fortran, OpenCL, Python, assembly, or any combination
- Get coarse-grained system data for an extended period or detailed results mapped to source code.
- Low system overhead

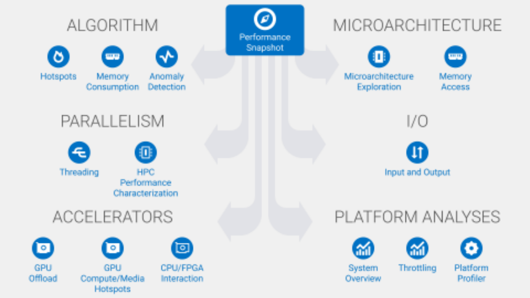

Figure 3: Intel VTune Profiler overview.

Intel Inspector

- Helps find memory errors and nondeterministic threading errors and problems
- Dynamic memory and threading error debugger for C, C++, and Fortran applications

Intel Trace Analyzer and Collector

- Profiles and analyzes MPI applications
- Discover temporal dependencies and bottlenecks
- Check the correctness of your application
- Locate potential programming errors, buffer overlaps, and deadlocks
- Visualize and understand parallel application behavior
- Evaluate profiling statistics and load balancing
- Analyze performance of subroutines or code blocks
- Learn about communication patterns, parameters, and performance data
- Identify communication hot spots
- Decrease time to solution and increase application efficiency

Components

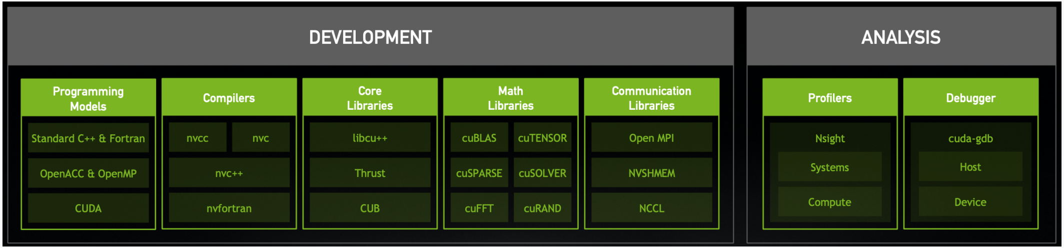

Figure 4: Components of the NVIDIA HPC SDK.

NVIDIA HPC SDK Compilers

Standards-based compilers based on the open-source LLVM compiler infrastructure.

nvc C11 compiler for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs

nvc++ C++17 compiler for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs

nvfortran Fortran 2003 compiler with many Fortran 2008 featutres implemented for NVIDIA GPUs and AMD, Intel, OpenPOWER, and Arm CPUs

nvcc CUDA C and CUDA C++ compiler driver for NVIDIA GPUs

Parallel Programming Models

C++ -stdpar C++ 17 Parallel Algorithms introduce parallel and vector concurrency through execution policies

OpenACC OpenACC directives for paralleliziation on CPUs and NVIDIA GPUs

OpenMP OpenMP directives for paralleliziation on CPUs and NVIDIA GPUs

CUDA C, C++, and Fortran varients of CUDA for parallelization on NVIDIA GPUs

Parallel Programming Models

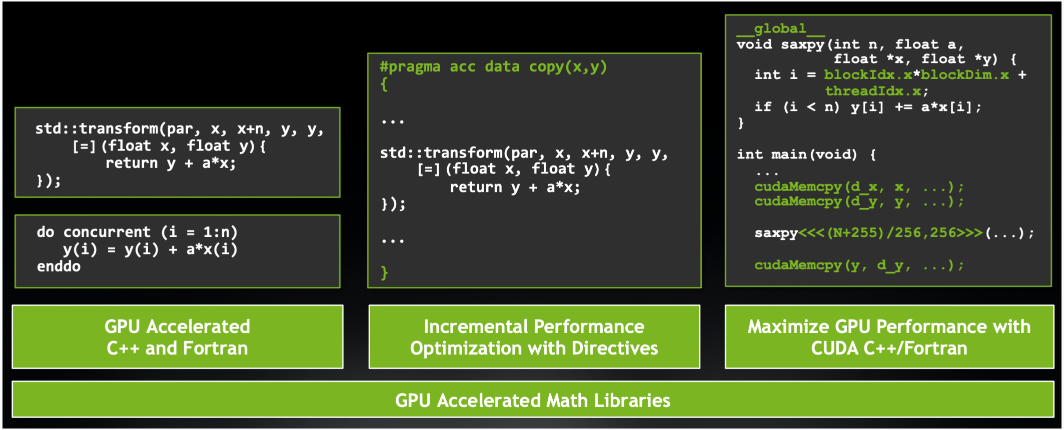

Figure 5: Parallel Programming Models

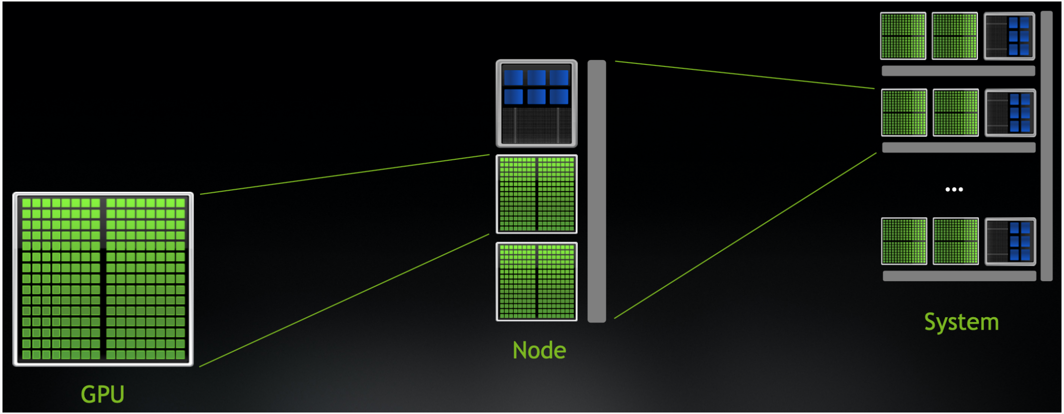

Figure 6: Parallel Programming Models

C++17

Parallel Algorithms
- In NVC++ 20.5
- Parallel and vector concurrency

Forward Progress Guarantees
- Extend the C++ execution model for accelerators.

Memory Model Clarifications
- Extend the C++ memory model for accelerators.

C++20

Scalable Synchronization Library
- Express thread synchronization that is portable and scalable across CPUs and accelerators.
- In libcu++ in CUDA 10.2:
  - `std::atomic<T>`
- In libcu++ in CUDA 11.0:
  - `std::barrier`
  - `std::counting_semaphore`
  - `std::atomic<T>::wait/notify_*`
- In libcu++ in the future:
  - `std::atomic_ref<T>`

C++23 and Beyond

Executors
- Simplify launching and managing parallel work across CPUs and accelerators.

`std::mdspan/mdarray`
- HPC-oriented multi-dimensional array abstractions.

Linear Algebra
- Standard C++ API to vendor BLAS libraries.

Extended Floating Point Types
- First-class support for formats new and old: `std::float16_t/float64_t`

Figure 7: Communications Libraries

C++ Standard Parallel Algorithms (stdpar)

```
std::sort(std::execution::par, c.begin(), c.end());

std::unique(std::execution::par, c.begin(), c.end());
```

- Introduced in C++17
- Parallel and vector concurrency via execution policies
  `std::execution::par, std::execution::par_seq, std::execution::seq`
- Several new algorithms in C++17 including
  - `std::for_each_n(POLICY, first, size, func)`
- Insert `std::execution::par` as first parameter when calling algorithms
- NVC++ 20.5: automatic GPU acceleration of C++17 parallel algorithms
  - Leverages CUDA Unified Memory

Figure 8: Communications Libraries

Tools

CUDA-GDB Debugging CUDA applications.

Nsight Compute Interactive kernel profiler for CUDA applications

Nsight System System-wide performance analysis tool designed to visualize application algorithms

Compute Sanitizer Functional correctness checking tools suite

NVTX API for annotating application events, code ranges, and resources for use with Nsight

Additional Resources

- C++ STL Algorithms Documentation
- NVIDIA HPC SDK Documentation
- C++ Concurrancy In Action, 2^{nd} Edition
- CppCon 2018: Bryce Adelstein Lelbach “The C++ Execution Model”
- C++17 Parallel Algorithms on NVIDIA GPUs with PGI C++

Additional Resources

- The oneAPI Specification
- Data Parallel C++ (Free eBook)
- Intel oneAPI Base Toolkit
- Intel oneAPI HPC Toolkit
- Intel oneAPI Toolkit Samples

Parallel Python

Standard Python Parallelization

- `threading`
- `multiprocessing`
- `asyncio`
- `queue`
- `concurrent.futures`
- Coroutines

Issues with Python Parallelization

- Coroutines (`asyncio`)
  - Minimal overhead
  - Subject to the GIL
- `threading`
  - Non-trivial overhead
  - Subject to the GIL
- `multiprocessing`
  - Substantial overhead

Python Global Interpreter Lock (GIL)

- Only one thread in a Python process can execute Python code at a time
- Calling certain libraries and functions release the GIL, *.e.g.* NumPy

```python
from threading import Thread

def fib(n):
    if n <= 2:
        return 1
    elif n == 0:
        return 0
    elif n < 0:
        raise Exception('fib(n) is undefined for n < 0')
    return fib(n - 1) + fib(n - 2)

if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument('-n', type=int, default=1)
    parser.add_argument('number', type=int, nargs='?', default=34)
    args = parser.parse_args()

    assert args.n >= 1, 'The number of threads has to be > 1'
    for i in range(args.n):
        t = Thread(target=fib, args=(args.number, ))
        t.start()
```

Multiple Processes

```python
import concurrent.futures as cf

def fib(n):
    if n <= 2:
        return 1
    elif n == 0:
        return 0
    elif n < 0:
        raise Exception('fib(n) is undefined for n < 0')
    return fib(n - 1) + fib(n - 2)

if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument('-n', type=int, default=1)
    parser.add_argument('number', type=int, nargs='?', default=34)
    args = parser.parse_args()

    assert args.n >= 1, 'The number of threads has to be > 1'
    with cf.ThreadPoolExecutor(max_workers=args.n) as pool:
        results = pool.map(fib, [args.number] * args.n)
```

Multiple Threads

```python
import concurrent.futures as cf

def fib(n):
    if n <= 2:
        return 1
    elif n == 0:
        return 0
    elif n < 0:
        raise Exception('fib(n) is undefined for n < 0')
    return fib(n - 1) + fib(n - 2)

if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument('-n', type=int, default=1)
    parser.add_argument('number', type=int, nargs='?', default=34)
    args = parser.parse_args()

    assert args.n >= 1, 'The number of threads has to be > 1'
    with cf.ProcessPoolExecutor(max_workers=args.n) as pool:
        results = pool.map(fib, [args.number] * args.n)
```

Multiprocess Queues

```python
import multiprocessing as mp

def fib(n):
    if n <= 2:
        return 1
    elif n == 0:
        return 0
    elif n < 0:
        raise Exception('fib(n) is undefined for n < 0')
    return fib(n - 1) + fib(n - 2)

def worker(inq, outq):
    while True:
        data = inq.get()
        if data is None:
            return
        fn, arg = data
        outq.put(fn(arg))
```

Multiprocess Queues

```python
if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument('-n', type=int, default=1)
    parser.add_argument('number', type=int, nargs='?', default=34)
    args = parser.parse_args()

    assert args.n >= 1, 'The number of threads has to be > 1'

    tasks = mp.Queue()
    results = mp.Queue()
    for i in range(args.n):
        tasks.put((fib, args.number))

    for i in range(args.n):
        mp.Process(target=worker, args=(tasks, results)).start()

    for i in range(args.n):
        print(results.get())

    for i in range(args.n):
        tasks.put(None)
```

Parallel Numba

- All Numba array operations
- Many NumPy array operations
- Many NumPy basic math functions
- Some NumPy BLAS functions

Parallel Numba

```python
from numba import njit, prange
import numpy as np

@njit(parallel=True, fastmath=True)
def two_d_array_reduction_prod(n):
    shp = (5000, 5000)
    result1 = 2 * np.ones(shp, np.int_)
    tmp = 2 * np.ones_like(result1)

    for i in prange(n):
        result1 *= tmp

    return result1

if __name__ == '__main__':
    import argparse

    parser = argparse.ArgumentParser()
    parser.add_argument('-n', type=int, default=1)
    args = parser.parse_args()

    two_d_array_reduction_prod(args.n)
```

Additional Resources

- Fluent Python
- High Performance Python
- In-Memory Analytics with Apache Arrow
- Productive and Efficient Data Science with Python
- Data Science with Python and Dask
- Distributed Computing with Python
- Parallel and High Performance Computing
- C++ High Performance - Second Edition

Assignments and Project

Assignments and Labs

- Complete and commit all assignments and labs by Tuesday, July 26 to receive grades
- All assignments are listed in the README
- Available for office hours by request and 30 minutes before and after Thursday's (07/21/22) session

Project

- Implement initial improvements for your three optimization targets to discuss at next Thursday's (07/28/22) session
- Available for office hours by request and 30 minutes before and after Thursday's (07/21/22) and Tuesday's (07/26/22) sessions
- Meet with each individual to discuss optimization targets