PGI® COMPILERS & TOOLS

C++17 PARALLEL ALGORITHMS ON NVIDIA GPUS WITH PGI C++

David Olsen

GTC S9770

March 20, 2019

NVIDIA

```
__global__ void saxpy_kernel(float* x, float* y, float* z, float a, int N) {
  for (int i = blockIdx.x * blockDim.x + threadIdx.x; i < N; i += gridDim.x * blockDim.x) {
    z[i] = x[i] * a + y[i];
  }
}
void saxpy(float* x, float* y, float* z, float a, int N) {
  size_t size = N * sizeof(float);
  float *d_x, *d_y, *d_z;
  cudaMalloc(&d_x, size);
  cudaMalloc(&d_y, size);
  cudaMalloc(&d_z, size);
  cudaMemcpy(d_x, x, size, cudaMemcpyHostToDevice);
  cudaMemcpy(d_y, y, size, cudaMemcpyHostToDevice);
  saxpy_kernel<<<64,256>>>(d_x, d_y, d_z, a, N);
  cudaMemcpy(z, d_z, size, cudaMemcpyDeviceToHost);
  cudaFree(d_x);
  cudaFree(d_y);
  cudaFree(d_z);
}
```

```
void saxpy(float* x, float* y, float* z, float a, int N) {
  for (int i = 0; i < N; ++i) {
    z[i] = x[i] * a + y[i];
  }
}
```

GPU C++ PROGRAMMING TODAY

OpenACC
More Science, Less Programming

OpenMP®
Enabling HPC since 1997

#pragmas

NVIDIA.
CUDA®

OpenCL

Language Extensions

Thrust

KOKKOS

Libraries

C++17 PARALLEL ALGORITHMS

Parallelism in Standard C++

Execution policies can be applied to most standard algorithms

```
std::execution::seq  = sequential
std::execution::par  = parallel
std::execution::par_unseq  = parallel + vectorized
```

Several existing algorithms were renamed

accumulate => reduce
inner_product => transform_reduce
partial_sum => inclusive_scan

C++17 PARALLEL ALGORITHMS

Example

C++98: `std::sort(c.begin(), c.end());`

C++17: `std::sort(std::execution::par, c.begin(), c.end());`

C++98:
```
double prod = std::accumulate(
              first, last, 1.0, std::multiplies());
```

C++17:
```
double prod = std::reduce(std::execution::par,
              first, last, 1.0, std::multiplies());
```

THE FUTURE OF GPU PROGRAMMING

Standard C++ | Directives | CUDA

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
```

```
#pragma acc data copy(x,y) {

...

std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});

...

}
```

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

GPU Accelerated Standard C++

Incremental Performance Optimization with OpenACC

Maximize GPU Performance with CUDA C++

PGI

NVIDIA

THE FUTURE OF GPU PROGRAMMING

Standard C++ | Directives | CUDA

Coming soon to a PGI C++ compiler near you

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});
```

```
#pragma acc data copy(x,y) {

...

std::transform(par, x, x+n, y, y,
    [=](float x, float y){
        return y + a*x;
});

...

}
```

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  ...
  cudaMemcpy(d_x, x, ...);
  cudaMemcpy(d_y, y, ...);

  saxpy<<<(N+255)/256,256>>>(...);

  cudaMemcpy(y, d_y, ...);
```

GPU Accelerated Standard C++

Incremental Performance Optimization with OpenACC

Maximize GPU Performance with CUDA C++

PGI — THE NVIDIA HPC SDK

- Fortran, C & C++ Compilers
  - Optimizing, SIMD Vectorizing, OpenMP
- Accelerated Computing Features
  - CUDA Fortran, OpenACC Directives
- Multi-Platform Solution
  - X86-64 and OpenPOWER Multicore CPUs
  - NVIDIA Tesla GPUs
  - Supported on Linux, macOS, Windows
- MPI/OpenMP/OpenACC Tools
  - Debugger
  - Performance Profiler
  - Interoperable with DDT, TotalView

PGI®

The Compilers & Tools
for Supercomputing

PGI Compilers, The NVIDIA HPC SDK: Updates for 2019

Michael Wolfe (NVIDIA, PGI Compiler Engineer)

Thursday, 10:00am, Room 211A

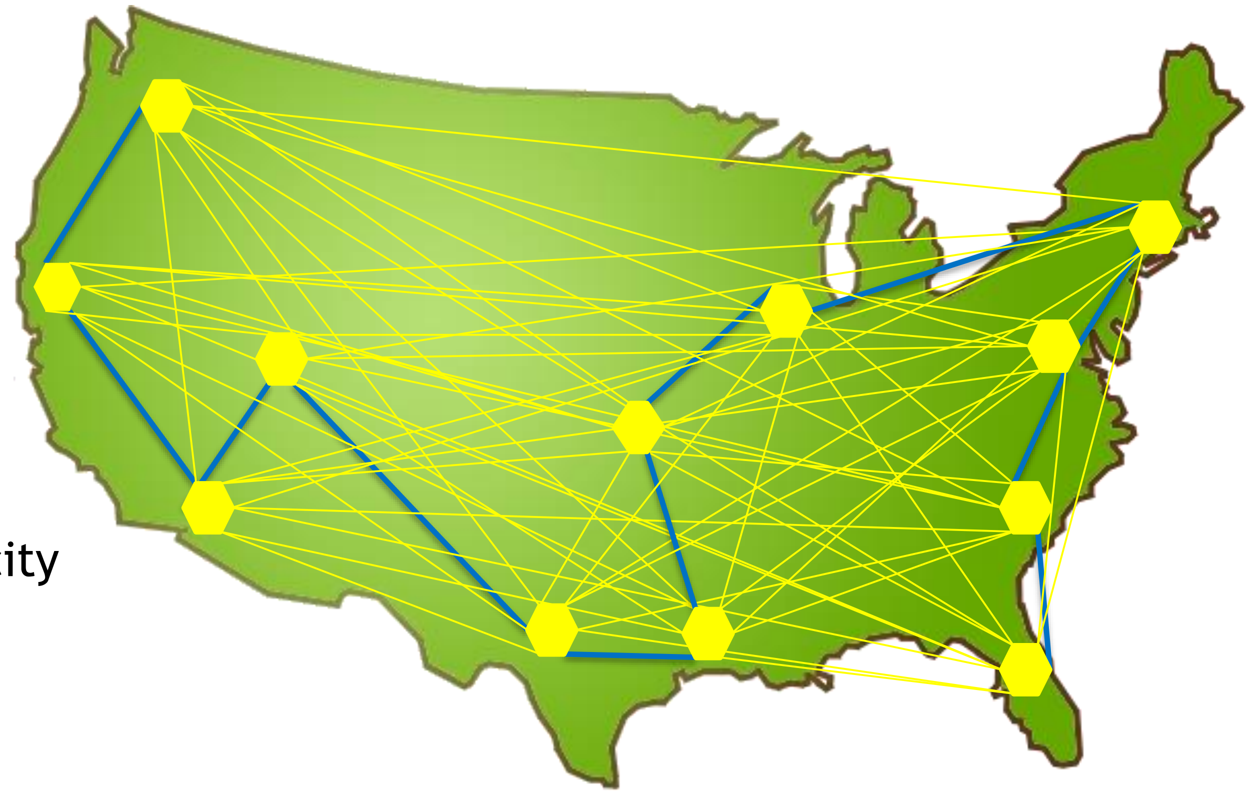
PGI®

The Compilers & Tools for Supercomputing

CODE EXAMPLES

TRAVELING SALESMAN

Find the shortest route that visits every city



TRAVELING SALESMAN

Sequential code

```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  for (long i = 0; i < num_routes; ++i) {
    int cost = 0;
    route_iterator it(i, N);
    int from = it.first();
    while (!it.done()) {
      int to = it.next();
      cost += distances[from*N + to];
      from = to;
    }
    best_route = route_cost::min(best_route, route_cost(i, cost));
  }
  return best_route;
}
```

TRAVELING SALESMAN

Helper code

`route_cost` is a (route ID, cost) pair, and a `min` function to return the least costly route

```
struct route_cost {
  long route;
  int cost;
  route_cost() : route(-1), cost(std::numeric_limits<int>::max()) { }
  route_cost(long route, int cost) : route(route), cost(cost) { }

  static route_cost min(route_cost const& x, route_cost const& y) {
    if (x.cost < y.cost) {
      return x;
    }
    return y;
  }
};
```

TRAVELING SALESMAN

Helper code

`Route_iterator` calculates a route, given a route ID and the number of cities

```
struct route_iterator {
  route_iterator(long route_id, int num_hops);
  bool done() const; // at the end of the route ?
  int first(); // first city of the route
  int next(); // next city of the route
};
```

TRAVELING SALESMAN

Sequential code

```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  for (long i = 0; i < num_routes; ++i) {
    int cost = 0;
    route_iterator it(i, N);
    int from = it.first();
    while (!it.done()) {
      int to = it.next();
      cost += distances[from*N + to];
      from = to;
    }
    best_route = route_cost::min(best_route, route_cost(i, cost));
  }
  return best_route;
}
```

TRAVELING SALESMAN

Sequential code

```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  for (long i = 0; i < num_routes; ++i) {
    int cost = 0;
    route_iterator it(i, N);
    int from = it.first();
    while (!it.done()) {
      int to = it.next();
      cost += distances[from*N + to];
      from = to;
    }
    best_route = route_cost::min(best_route, route_cost(i, cost));
  }
  return best_route;
}
```

TRAVELING SALESMAN

Sequential code

```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  for (long i = 0; i < num_routes; ++i) {
    int cost = 0;
    route_iterator it(i, N);
    int from = it.first();
    while (!it.done()) {
      int to = it.next();
      cost += distances[from*N + to];
      from = to;
    }
    best_route = route_cost::min(best_route, route_cost(i, cost));
  }
  return best_route;
}
```

TRAVELING SALESMAN

Analysis

```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  for (long i = 0; i < num_routes; ++i) {
    int cost = 0;
    route_iterator it(i, N);
    int from = it.first();
    while (!it.done()) {
      int to = it.next();
      cost += distances[from*N + to];
      from = to;
    }
    best_route = route_cost::min(best_route, route_cost(i, cost));
  }
  return best_route;
}
```

TRAVELING SALESMAN

Manual threading

```
route_cost find_best_route(int const* distances,
                           int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  std::mutex route_mutex;
  int num_threads =
        std::thread::hardware_concurrency();
  if (num_threads == 0) num_threads = 4;
  std::vector<std::thread> threads;
  threads.reserve(num_threads);
  for (int t = 0; t < num_threads; ++t) {
    threads.push_back(std::thread(
      [=, &best_route, &route_mutex](int chunk) {
        route_cost local_best;
        for (long i = chunk; i < num_routes;
             i += num_threads) {
          int cost = 0;
          route_iterator it(i, N);
          int from = it.first();
          while (!it.done()) {
            int to = it.next();
            cost += distances[from*N + to];
            from = to;
          }
          local_best = route_cost::min(
                    local_best, route_cost(i, cost));
        }
        std::lock_guard<std::mutex> lck(route_mutex);
        best_route = route_cost::min(
                         best_route, local_best);
      }, t));
  }
  for (std::thread& th : threads) {
    th.join();
  }
  return best_route;
}
```

TRAVELING SALESMAN

Manual threading

```
route_cost find_best_route(int const* distances,
                           int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  std::mutex route_mutex;
  int num_threads =
        std::thread::hardware_concurrency();
  if (num_threads == 0) num_threads = 4;
  std::vector<std::thread> threads;
  threads.reserve(num_threads);
  for (int t = 0; t < num_threads; ++t) {
    threads.push_back(std::thread(
      [=, &best_route, &route_mutex](int chunk) {
        route_cost local_best;
        for (long i = chunk; i < num_routes;
             i += num_threads) {
          int cost = 0;
          route_iterator it(i, N);
          int from = it.first();
          while (!it.done()) {
            int to = it.next();
            cost += distances[from*N + to];
            from = to;
          }
          local_best = route_cost::min(
                    local_best, route_cost(i, cost));
        }
        std::lock_guard<std::mutex> lck(route_mutex);
        best_route = route_cost::min(
                               best_route, local_best);
      }, t));
  }
  for (std::thread& th : threads) {
    th.join();
  }
  return best_route;
}
```

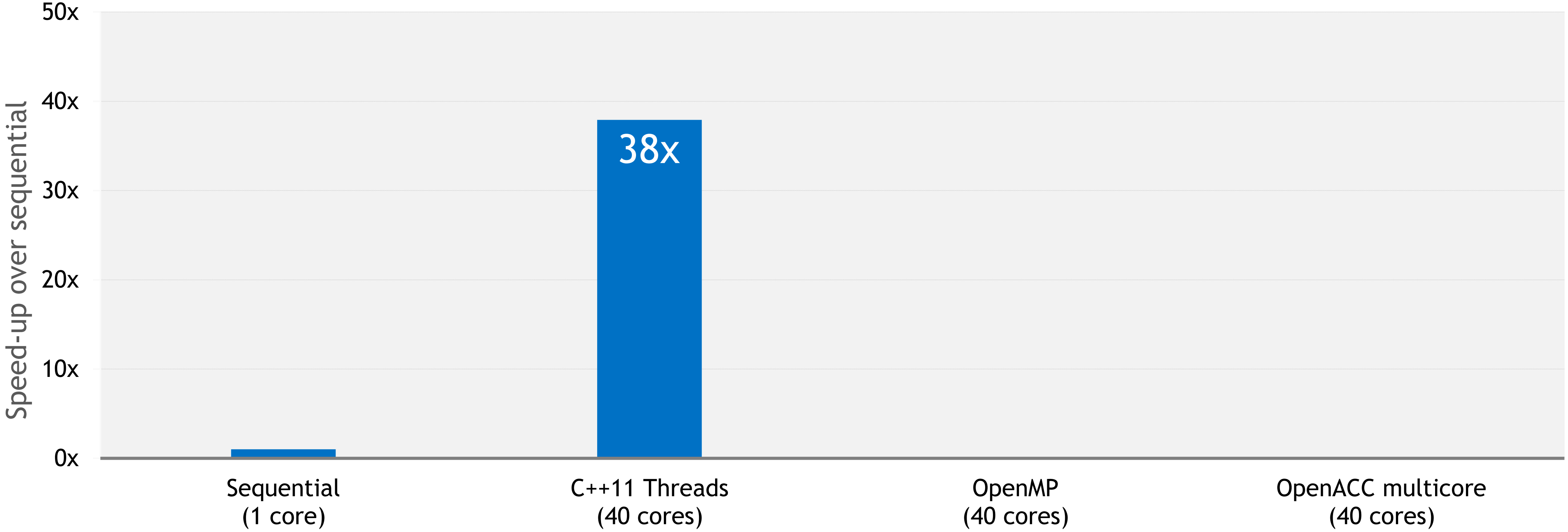
TRAVELING SALESMAN

Manual threading

```
route_cost find_best_route(int const* distances,
                           int N) {
  long num_routes = factorial(N);
  route_cost best_route;
  std::mutex route_mutex;
  int num_threads =
        std::thread::hardware_concurrency();
  if (num_threads == 0) num_threads = 4;
  std::vector<std::thread> threads;
  threads.reserve(num_threads);
  for (int t = 0; t < num_threads; ++t) {
    threads.push_back(std::thread(
      [=, &best_route, &route_mutex](int chunk) {
        route_cost local_best;
        for (long i = chunk; i < num_routes;
             i += num_threads) {
          int cost = 0;
          route_iterator it(i, N);
          int from = it.first();
          while (!it.done()) {
            int to = it.next();
            cost += distances[from*N + to];
            from = to;
          }
          local_best = route_cost::min(
                    local_best, route_cost(i, cost));
        }
        std::lock_guard<std::mutex> lck(route_mutex);
        best_route = route_cost::min(
                              best_route, local_best);
      }, t));
  }
  for (std::thread& th : threads) {
    th.join();
  }
  return best_route;
}
```

TRAVELING SALESMAN PERFORMANCE

Dual-socket Xeon Gold 6148 server 2.3 GHz



System: Dual-socket Intel Xeon Gold 6148 with a total of 40 physical cores and 1 Tesla V100-PCIE-16GB GPU

Compilers and options:
Sequential: g++ -O3
Threads: g++ -O3 -pthread

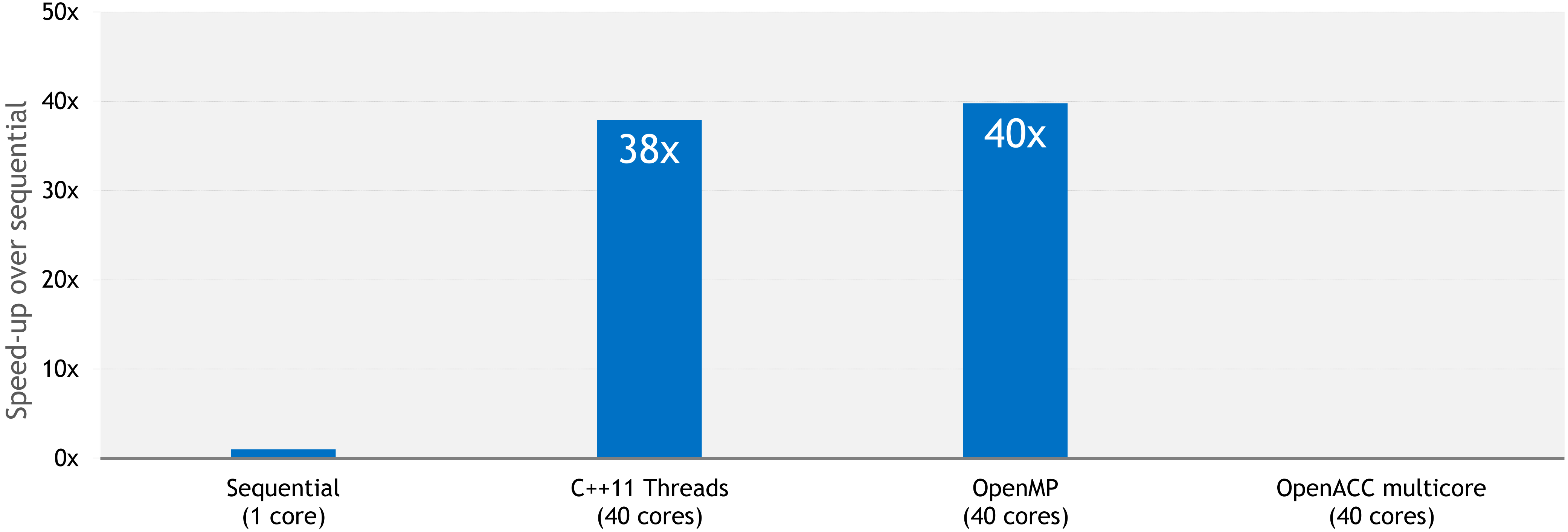
TRAVELING SALESMAN

OpenMP

```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  #pragma omp declare reduction \
    (route_min : route_cost : omp_out = route_cost::min_func(omp_out, omp_in)) \
    initializer(omp_priv = route_cost())
  route_cost best_route;
  #pragma omp parallel for reduction(route_min : best_route)
  for (long i = 0; i < num_routes; ++i) {
    int cost = 0;
    route_iterator it(i, N);
    int from = it.first();
    while (!it.done()) {
      int to = it.next();
      cost += distances[from*N + to];
      from = to;
    }
    best_route = route_cost::min(best_route, route_cost(i, cost));
  }
  return best_route;
}
```

TRAVELING SALESMAN PERFORMANCE

Dual-socket Xeon Gold 6148 server 2.3 GHz



System: Dual-socket Intel Xeon Gold 6148 with a total of 40 physical cores and 1 Tesla V100-PCIE-16GB GPU

Compilers and options:
Sequential: g++ -O3
Threads: g++ -O3 -pthread
OpenMP: g++ -O3 -fopenmp

PGI

NVIDIA

TRAVELING SALESMAN

OpenACC

```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  int num_blocks = 1024;
  int block_size = 128;
  int threads = num_blocks * block_size;
  route_cost* best_thread = new route_cost[threads];
  #pragma acc enter data copyin(best_thread[0:threads]) \
              copyin(distances[0:N*N])
  #pragma acc parallel num_gangs(num_blocks) \
              vector_length(block_size) \
              present(best_thread, distances)
  #pragma acc loop gang
  for (int ig = 0; ig < num_blocks; ++ig) {
    #pragma acc loop vector
    for (int iv = 0; iv < block_size; ++iv) {
      route_cost best_route;
      int idx = ig * block_size + iv;
      #pragma acc loop seq
      for (long i = idx; i < num_routes; i+=threads) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        best_route = route_cost::min(best_route,
                                     route_cost(i, cost));
      }
      best_thread[idx] = best_route;
    }
  }
  #pragma acc update self(best_thread[:threads])
  route_cost best_route;
  for (long i = 0; i < threads; ++i) {
    best_route = route_cost::min(best_route,
                                 best_thread[i]);
  }
  #pragma acc exit data delete(best_thread,distances)
  delete[] best_thread;
  return best_route;
}
```

PGI

NVIDIA

TRAVELING SALESMAN

OpenACC

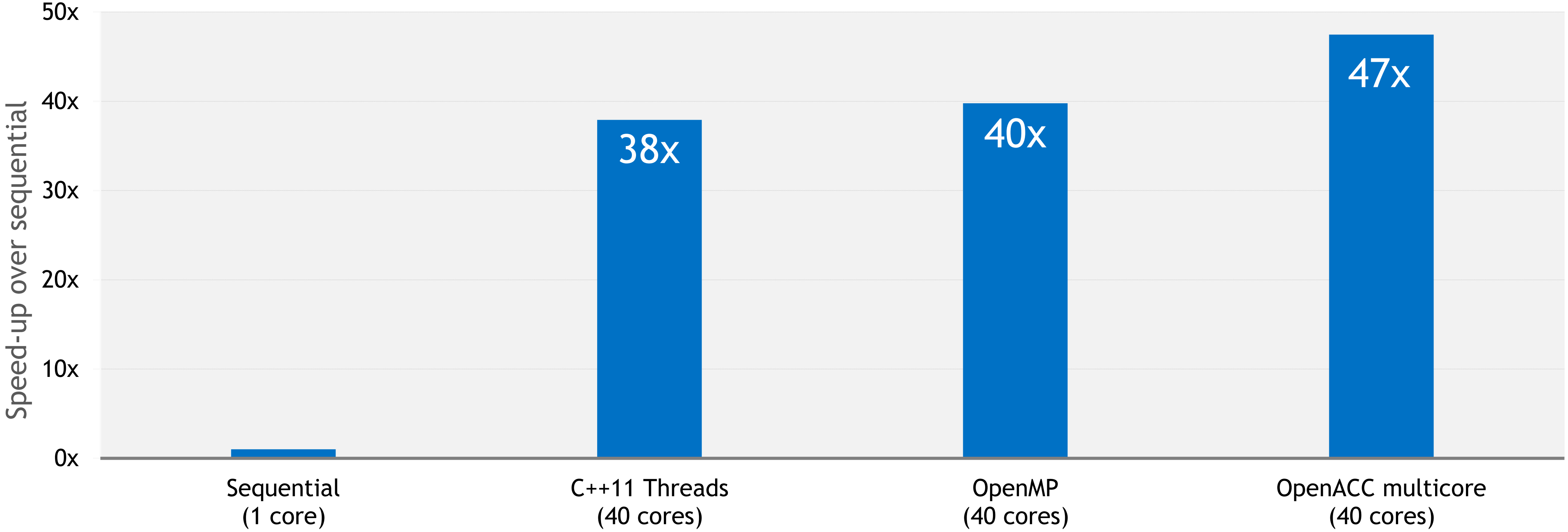
```
route_cost find_best_route(int const* distances, int N) {
  long num_routes = factorial(N);
  int num_blocks = 1024;
  int block_size = 128;
  int threads = num_blocks * block_size;
  route_cost* best_thread = new route_cost[threads];
  #pragma acc enter data copyin(best_thread[0:threads]) \
              copyin(distances[0:N*N])
  #pragma acc parallel num_gangs(num_blocks) \
              vector_length(block_size) \
              present(best_thread, distances)
  #pragma acc loop gang
  for (int ig = 0; ig < num_blocks; ++ig) {
    #pragma acc loop vector
    for (int iv = 0; iv < block_size; ++iv) {
      route_cost best_route;
      int idx = ig * block_size + iv;
      #pragma acc loop seq
      for (long i = idx; i < num_routes; i+=threads) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        best_route = route_cost::min(best_route,
                                     route_cost(i, cost));
      }
      best_thread[idx] = best_route;
    }
  }
  #pragma acc update self(best_thread[:threads])
  route_cost best_route;
  for (long i = 0; i < threads; ++i) {
    best_route = route_cost::min(best_route,
                                 best_thread[i]);
  }
  #pragma acc exit data delete(best_thread,distances)
  delete[] best_thread;
  return best_route;
}
```

PGI

NVIDIA

TRAVELING SALESMAN PERFORMANCE

Dual-socket Xeon Gold 6148 server 2.3 GHz



System: Dual-socket Intel Xeon Gold 6148 with a total of 40 physical cores and 1 Tesla V100-PCIE-16GB GPU

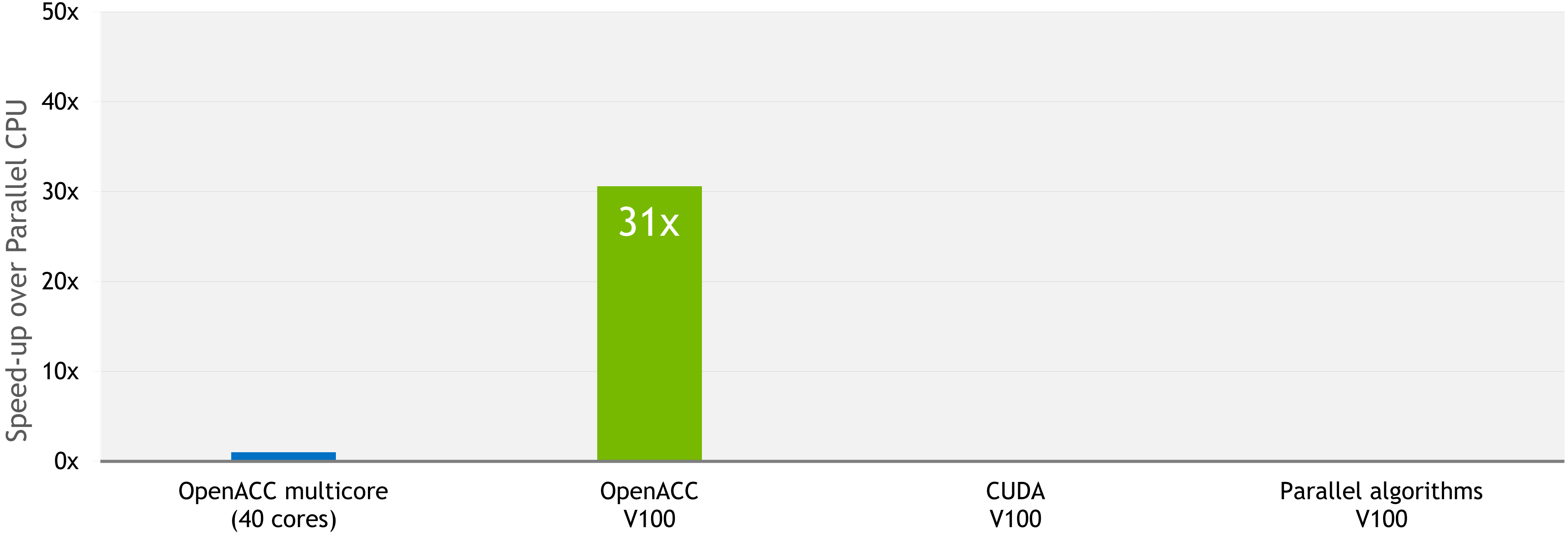
Compilers and options:
Sequential: g++ -O3
Threads: g++ -O3 -pthread
OpenMP: g++ -O3 -fopenmp
OpenACC: pgc++ -fast -ta=multicore

PGI

NVIDIA

TRAVELING SALESMAN PERFORMANCE

Dual-socket Xeon Gold vs 1x Tesla V100



System: Dual-socket Intel Xeon Gold 6148 with a total of 40 physical cores and 1 Tesla V100-PCIE-16GB GPU

Compilers and options:
OpenACC: pgc++ -fast -ta=multicore
OpenACC GPU: PGI 19.1 pgc++ -fast -acc

TRAVELING SALESMAN

CUDA

```
route_cost find_best_route(int const* distances, int N) {
  int* dev_distances;
  cudaMalloc(&dev_distances, N * N * sizeof(int));
  cudaMemcpy(dev_distances, distances, N * N * sizeof(float),
             cudaMemcpyHostToDevice);
  long num_routes = factorial(N);
  int threads = 1024;
  int blocks = std::min((num_routes + threads - 1) / threads, 1024L);
  route_cost* block_best;
  cudaMalloc(&block_best, blocks * sizeof(route_cost));
  find_best_kernel<<<blocks, threads>>>(dev_distances, N, num_routes,
                                        block_best);
  cudaDeviceSynchronize();
  route_cost* host_block_best = new route_cost[blocks];
  cudaMemcpy(host_block_best, block_best, blocks * sizeof(route_cost),
             cudaMemcpyDeviceToHost);
  route_cost best_route;
  for (int i = 0; i < blocks; ++i) {
    best_route = route_cost::min(best_route, host_block_best[i]);
  }
  cudaFree(block_best);
  cudaFree(dev_distances);
  delete[] host_block_best;
  return best_route;
}
```

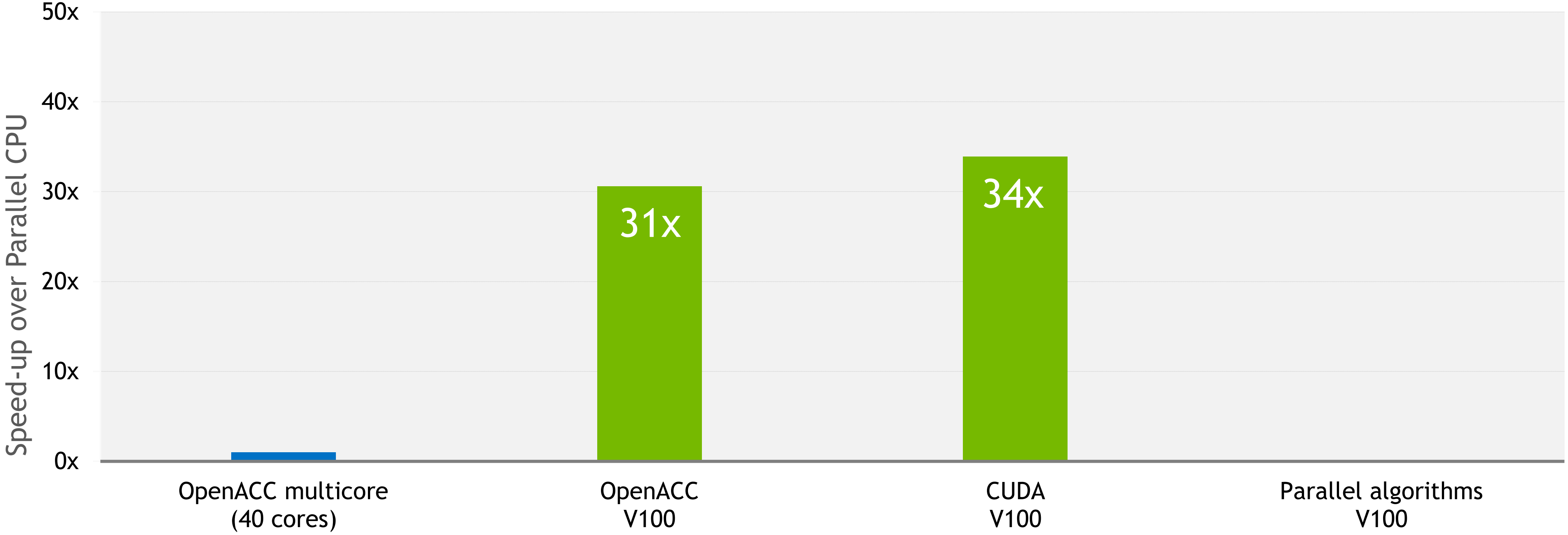
TRAVELING SALESMAN

CUDA

```
__global__ void find_best_kernel(int* distances, int N, long num_routes, route_cost* block_best) {
  static __shared__ route_cost warp_best[32];
  route_cost local_best;
  for (long i = blockIdx.x * blockDim.x + threadIdx.x; i < num_routes; i += blockDim.x * gridDim.x) {
    int cost = 0;
    route_iterator it(i, N);
    int from = it.first();
    while (!it.done()) {
      int to = it.next();
      cost += distances[from*N + to];
      from = to;
    }
    local_best = route_cost::min(local_best, route_cost(i, cost));
  }
  int lane = threadIdx.x % warpSize;
  int warpId = threadIdx.x / warpSize;
  for (int offset = warpSize / 2; offset > 0; offset /= 2)
    local_best = route_cost::min(local_best, route_cost(__shfl_down_sync((unsigned)-1, local_best.route, offset),
                                   __shfl_down_sync((unsigned)-1, local_best.cost, offset)));
  if (lane == 0) warp_best[warpId] = local_best;
  __syncthreads();
  if (warpId == 0) {
    local_best = warp_best[lane];
    for (int offset = warpSize / 2; offset > 0; offset /= 2)
      local_best = route_cost::min(local_best, route_cost(__shfl_down_sync((unsigned)-1, local_best.route, offset),
                                     __shfl_down_sync((unsigned)-1, local_best.cost, offset)));
    if (lane == 0) block_best[blockIdx.x] = local_best;
  }
}
```

TRAVELING SALESMAN PERFORMANCE

Dual-socket Xeon Gold vs 1x Tesla V100



System: Dual-socket Intel Xeon Gold 6148 with a total of 40 physical cores and 1 Tesla V100-PCIE-16GB GPU

Compilers and options:
OpenACC: pgc++ -fast -ta=multicore
OpenACC GPU: PGI 19.1 pgc++ -fast -acc
CUDA: CUDA 10.0 nvcc -O3

TRAVELING SALESMAN

Parallel algorithm

```
route_cost find_best_route(int const* distances, int N) {
  return std::transform_reduce(std::execution::par,
      counting_iterator<long>(0L), counting_iterator<long>(factorial(N)),
      route_cost(),
      route_cost::min,
      [=](long i) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        return route_cost(i, cost);
      });
}
```

TRAVELING SALESMAN

Parallel algorithm

```
route_cost find_best_route(int const* distances, int N) {
  return std::transform_reduce(std::execution::par,
      counting_iterator<long>(0L), counting_iterator<long>(factorial(N)),
      route_cost(),
      route_cost::min,
      [=](long i) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        return route_cost(i, cost);
      });
}
```

TRAVELING SALESMAN

Parallel algorithm

```
route_cost find_best_route(int const* distances, int N) {
  return std::transform_reduce(std::execution::par,
      counting_iterator<long>(0L), counting_iterator<long>(factorial(N)),
      route_cost(),
      route_cost::min,
      [=](long i) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        return route_cost(i, cost);
      });
}
```

PGI

NVIDIA

TRAVELING SALESMAN

Parallel algorithm

```
route_cost find_best_route(int const* distances, int N) {
  return std::transform_reduce(std::execution::par,
      counting_iterator<long>(0L), counting_iterator<long>(factorial(N)),
      route_cost(),
      route_cost::min,
      [=](long i) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        return route_cost(i, cost);
      });
}
```

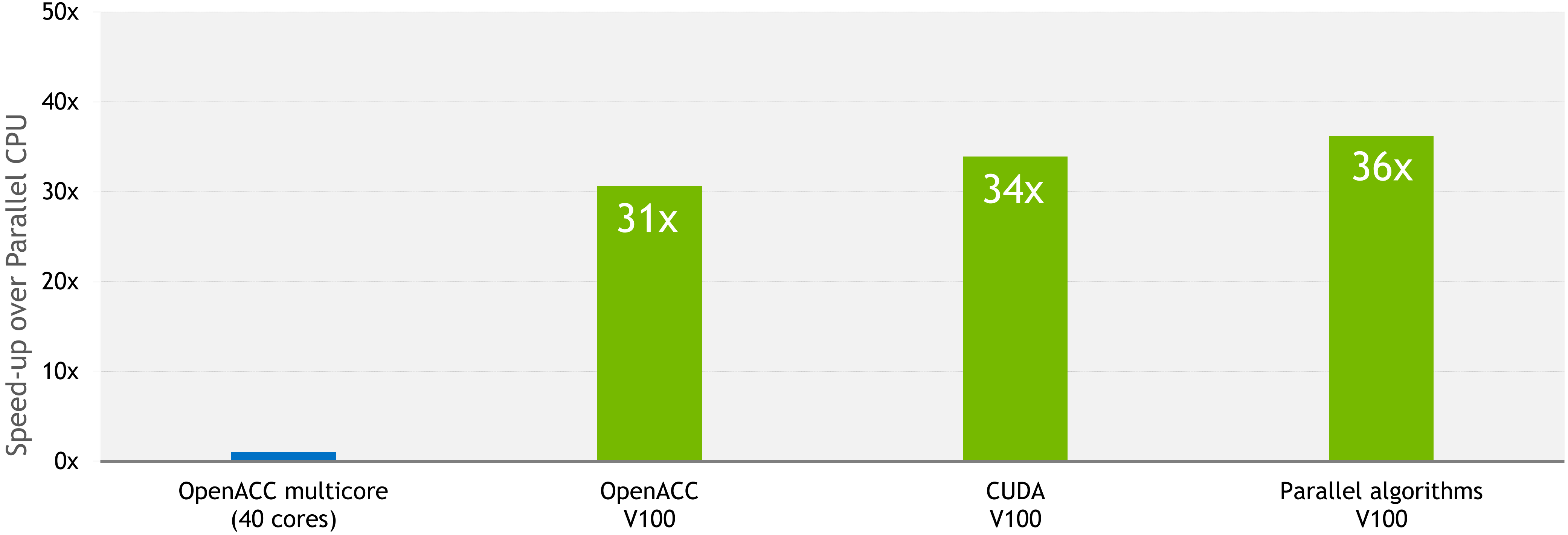
TRAVELING SALESMAN

Parallel algorithm

```
route_cost find_best_route(int const* distances, int N) {
  return std::transform_reduce(std::execution::par,
      counting_iterator<long>(0L), counting_iterator<long>(factorial(N)),
      route_cost(),
      route_cost::min,
      [=](long i) {
        int cost = 0;
        route_iterator it(i, N);
        int from = it.first();
        while (!it.done()) {
          int to = it.next();
          cost += distances[from*N + to];
          from = to;
        }
        return route_cost(i, cost);
      });
}
```

TRAVELING SALESMAN PERFORMANCE

Dual-socket Xeon Gold vs 1x Tesla V100

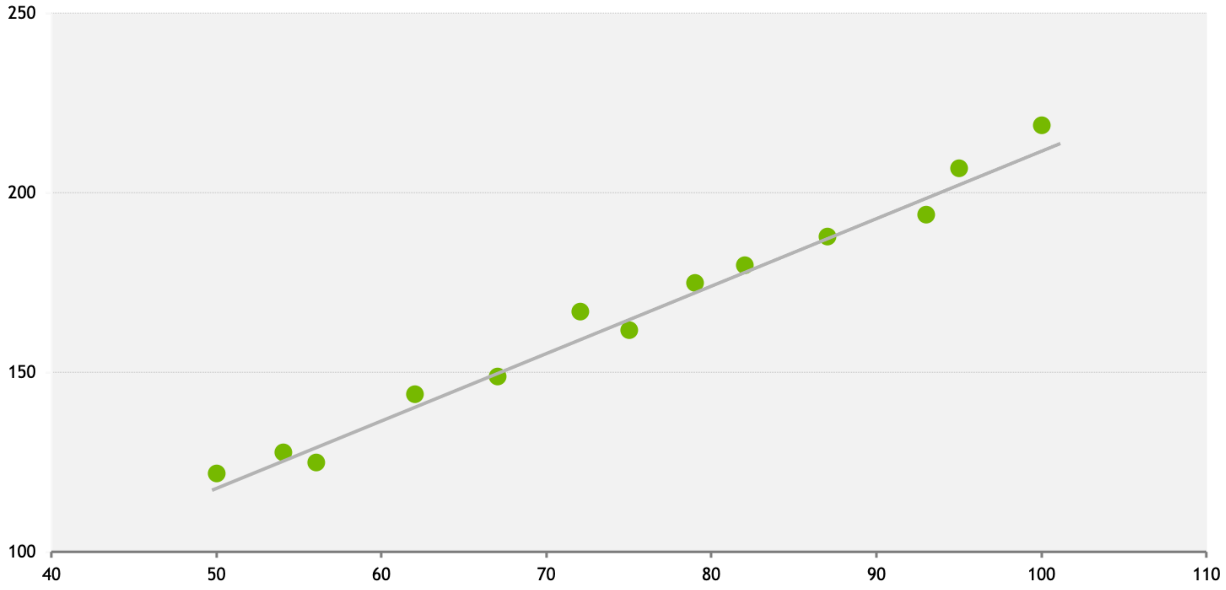


System: Dual-socket Intel Xeon Gold 6148 with a total of 40 physical cores and 1 Tesla V100-PCIE-16GB GPU

Compilers and options:
OpenACC: pgc++ -fast -ta=multicore
OpenACC GPU: PGI 19.1 pgc++ -fast -acc
CUDA: CUDA 10.0 nvcc -O3
pSTL transform_reduce(): PGI development

LINEAR ANALYSIS

Given two data sets, calculate the coefficient of correlation between them, along with the slope and intercept



LINEAR ANALYSIS

Sequential code

```
relation calculate_relation(int N, float const* x, float const* y) {
  relation result;
  float xm = std::accumulate(x, x + N, 0.0f) / N;
  float ym = std::accumulate(y, y + N, 0.0f) / N;
  float covariance = std::inner_product(x, x + N, y, 0.0f, std::plus<float>(),
       [=](float xi, float yi) { return (xi - xm) * (yi - ym); });
  float x_variance = std::accumulate(x, x + N, 0.0f,
       [=](float sum, float xi) { return sum + (xi - xm) * (xi - xm); });
  float y_variance = std::accumulate(y, y + N, 0.0f,
       [=](float sum, float yi) { return sum + (yi - ym) * (yi - ym); });
  result.correlation = covariance / std::sqrt(x_variance * y_variance);
  result.slope = covariance / x_variance;
  result.intercept = ym - result.slope * xm;
  return result;
}
```

LINEAR ANALYSIS

Sequential code

```
relation calculate_relation(int N, float const* x, float const* y) {
  relation result;
  float xm = std::accumulate(x, x + N, 0.0f) / N;
  float ym = std::accumulate(y, y + N, 0.0f) / N;
  float covariance = std::inner_product(x, x + N, y, 0.0f, std::plus<float>(),
       [=](float xi, float yi) { return (xi - xm) * (yi - ym); });
  float x_variance = std::accumulate(x, x + N, 0.0f,
       [=](float sum, float xi) { return sum + (xi - xm) * (xi - xm); });
  float y_variance = std::accumulate(y, y + N, 0.0f,
       [=](float sum, float yi) { return sum + (yi - ym) * (yi - ym); });
  result.correlation = covariance / std::sqrt(x_variance * y_variance);
  result.slope = covariance / x_variance;
  result.intercept = ym - result.slope * xm;
  return result;
}
```

LINEAR ANALYSIS

Sequential code

```
relation calculate_relation(int N, float const* x, float const* y) {
  relation result;
  float xm = std::accumulate(x, x + N, 0.0f) / N;
  float ym = std::accumulate(y, y + N, 0.0f) / N;
  float covariance = std::inner_product(x, x + N, y, 0.0f, std::plus<float>(),
       [=](float xi, float yi) { return (xi - xm) * (yi - ym); });
  float x_variance = std::accumulate(x, x + N, 0.0f,
       [=](float sum, float xi) { return sum + (xi - xm) * (xi - xm); });
  float y_variance = std::accumulate(y, y + N, 0.0f,
       [=](float sum, float yi) { return sum + (yi - ym) * (yi - ym); });
  result.correlation = covariance / std::sqrt(x_variance * y_variance);
  result.slope = covariance / x_variance;
  result.intercept = ym - result.slope * xm;
  return result;
}
```

LINEAR ANALYSIS

OpenMP

```
relation calculate_relation(int N, float const* x, float const* y) {
  relation result;
  float xm = 0.0f, ym = 0.0f;
  #pragma omp parallel for reduction(+: xm, ym)
  for (int i = 0; i < N; ++i) {
    xm += x[i]; ym += y[i];
  }
  xm /= N; ym /= N;
  float covariance = 0.0f, x_variance = 0.0f, y_variance = 0.0f;
  #pragma omp parallel for reduction(+: covariance, x_variance, y_variance)
  for (int i = 0; i < N; ++i) {
    float xd = x[i] - xm, yd = y[i] - ym;
    covariance += xd * yd;
    x_variance += xd * xd;
    y_variance += yd * yd;
  }
  result.correlation = covariance / std::sqrt(x_variance * y_variance);
  result.slope = covariance / x_variance;
  result.intercept = ym - result.slope * xm;
  return result;
}
```

LINEAR ANALYSIS

OpenACC

```
relation calculate_relation(int N, float const* x, float const* y) {
  relation result;
  #pragma acc data present(x[0:N], y[0:N])
  {
    float xm = 0.0f, ym = 0.0f;
    #pragma acc parallel loop reduction(+: xm, ym)
    for (int i = 0; i < N; ++i) {
      xm += x[i]; ym += y[i];
    }
    xm /= N; ym /= N;
    float covariance = 0.0f, x_variance = 0.0f, y_variance = 0.0f;
    #pragma acc parallel loop reduction(+: covariance, x_variance, y_variance)
    for (int i = 0; i < N; ++i) {
      float xd = x[i] - xm, yd = y[i] - ym;
      covariance += xd * yd;   x_variance += xd * xd;   y_variance += yd * yd;
    }
    result.correlation = covariance / std::sqrt(x_variance * y_variance);
    result.slope = covariance / x_variance;
    result.intercept = ym - result.slope * xm;
  }
  return result;
}
```

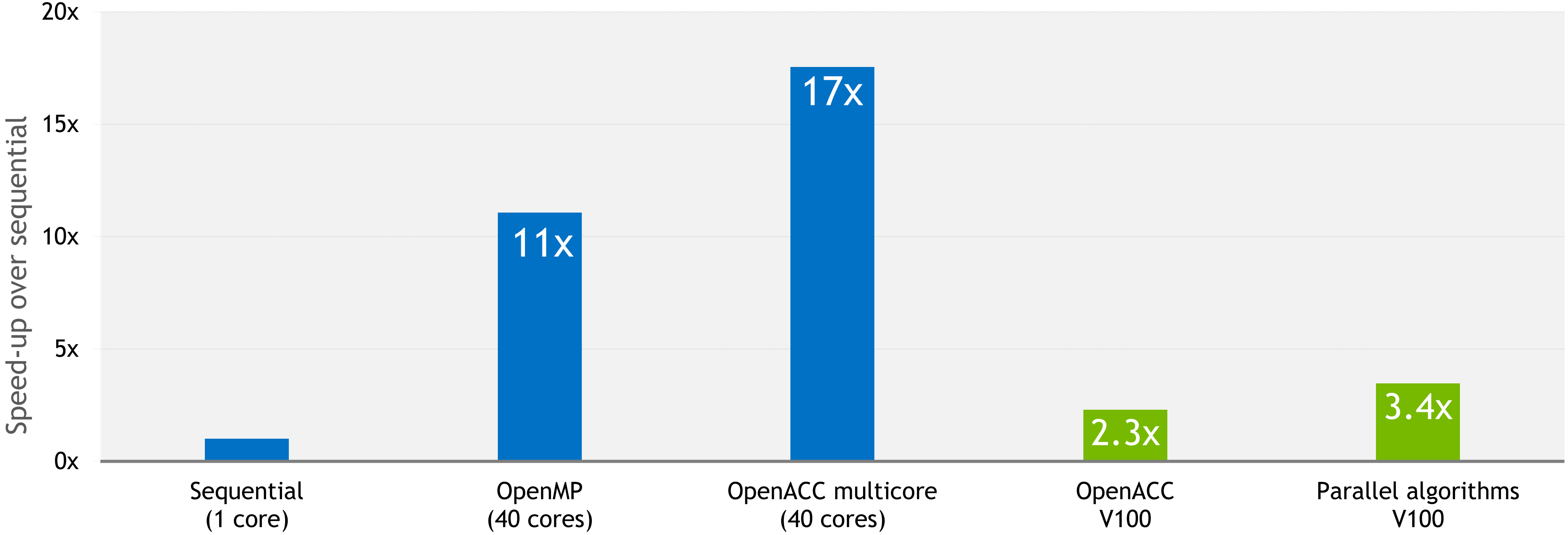
LINEAR ANALYSIS

Parallel algorithm

```
relation calculate_relation(int N, float const* x, float const* y) {
  relation result;
  float xm = std::reduce(std::execution::par, x, x + N) / N;
  float ym = std::reduce(std::execution::par, y, y + N) / N;
  float covariance = std::transform_reduce(std::execution::par,
      x, x + N, y, 0.0f, std::plus<float>(),
      [=](float xi, float yi) { return (xi - xm) * (yi - ym); });
  float x_variance = std::transform_reduce(std::execution::par,
      x, x + N, 0.0f, std::plus<float>(),
      [=](float xi) { return (xi - xm) * (xi - xm); });
  float y_variance = std::transform_reduce(std::execution::par,
      y, y + N, 0.0f, std::plus<float>(),
      [=](float yi) { return (yi - ym) * (yi - ym); });
  result.correlation = covariance / std::sqrt(x_variance * y_variance);
  result.slope = covariance / x_variance;
  result.intercept = ym - result.slope * xm;
  return result;
}
```

LINEAR ANALYSIS PERFORMANCE

Dual-socket Xeon Gold vs 1x Tesla V100



System: Dual-socket Intel Xeon Gold 6148 with a total of 40 physical cores and 1 Tesla V100-PCIE-16GB GPU

Compilers and options:
Sequential: g++ -O3
OpenMP: g++ -O3 -fopenmp
OpenACC CPU: pgc++ -fast -ta=multicore
OpenACC V100: pgc++ -fast -acc
pSTL transform_reduce(): PGI development

PGI

NVIDIA

OTHER ALGORITHMS

Parallel algorithm on GPU vs. sequential algorithm on CPU:

`transform`:	1,160 x
`for_each`:	1,054 x
`transform_reduce`:	458 x
`adjacent_difference`:	457 x
`reduce`:	280 x
`sort`:	46 x

LIMITATIONS

FUNCTION POINTERS

Don't pass function pointers to algorithms that will run on the GPU

```
void square(int& x) { x = x * x; }

//...

std::for_each(std::execution::par, v.begin(), v.end(),
      &square); // Fails: uses raw function pointer
```

FUNCTION POINTERS

Use function objects or lambdas instead

```
struct square {
      void operator()(int& x) const { x = x * x; }
};
//...
std::for_each(std::execution::par, v.begin(), v.end(),
      square()); // OK, function object

std::for_each(std::execution::par, v.begin(), v.end(),
      [](int& x) { x = x * x; }); // OK, lambda
```

FUNCTION POINTERS

Function calls can be wrapped in a lambda if necessary

```
void big_function(int& x) {
      // ... lots of code ...
}

// ...

std::for_each(std::execution::par, v.begin(), v.end(),
      [](int& x) { big_function(x); }); // OK, no function pointer
```

MEMORY ISSUES

History

CPU and GPU have different address spaces

Data needed to be explicitly copied between CPU memory and GPU memory

A lot of effort and code was spent managing data movement

MEMORY ISSUES

Unified memory

Trend is toward a shared virtual address space

Data is moved automatically by the OS and drivers between CPU and GPU

Not all the way there yet…

MEMORY ISSUES

Unified Memory

Current state of the PGI C++ parallel algorithms implementation:

Heap memory is automatically shared between CPU and GPU

Stack memory and global memory are not shared

MEMORY ISSUES

Heap only

All pointers used in parallel algorithms must point to the heap

```
std::vector<int> v = ...;
std::sort(std::execution::par,
      v.begin(), v.end()); // OK, vector allocates on heap
```

```
std::array<int, 1024> a = ...;
std::sort(std::execution::par,
      a.begin(), a.end()); // Fails, array stored on the stack
```

MEMORY ISSUES

Some pointers to the stack are hard to see

```
void saxpy(float* x, float* y, int N, float a) {

  std::transform(std::execution::par, x, x + N, y, y,
      [&](float xi, float yi) { return xi * a + yi; });
}
```

PGI

NVIDIA

MEMORY ISSUES

Lambda Captures

Some pointers to the stack are hard to see

```
void saxpy(float* x, float* y, int N, float a) {

  std::transform(std::execution::par, x, x + N, y, y,
      [&](float xi, float yi) { return xi * a + yi; });
}
```

Capture-by-reference often results in a reference to the stack

MEMORY ISSUES

Lambda Captures

Some pointers to the stack are hard to see

```
void saxpy(float* x, float* y, int N, float a) {

  std::transform(std::execution::par, x, x + N, y, y,
      [=](float xi, float yi) { return xi * a + yi; });
}
```

Capture-by-reference often results in a reference to the stack

Capture-by-value works better, because there is no hidden reference

OTHER LIMITATIONS

GPU code does not have access to the operating system or pre-compiled standard library

Usually works:

- template classes and functions
- inlined functions
- math functions

Usually doesn't work:

- non-template library functions
- OS functions

CONCLUSION

C++17 parallel algorithms running on GPUs with the PGI C++ compiler

- Linux x86 and Linux OpenPOWER with NVIDIA GPUs
- Tech preview in the 2nd half of 2019
- Available in the 1st half of 2020

NVIDIA®