GPU PROGRAMMING WITH STANDARD C++17

David Olsen | PGI C++ compiler team at NVIDIA

SC19 | Denver | November 21, 2019

GPU C++ PROGRAMMING TODAY

#pragmas

Language Extensions

Libraries

C++17 PARALLEL ALGORITHMS

Parallelism in Standard C++

C++17 standardizes running algorithms in parallel, including on GPUs

Insert `std::execution::par` as first parameter when calling algorithms

Examples:

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

double product = std::reduce(std::execution::par,
    first, last, 1.0, std::multiplies<double>());
```


NVIDIA

THE FUTURE OF GPU C++ PROGRAMMING

Standard C++ | Directives | CUDA

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

GPU Accelerated Standard C++

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

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

}
```

Incremental Performance Optimization with OpenACC

```
__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, ...);
```

Maximize GPU Performance with CUDA C++

THE FUTURE OF GPU C++ PROGRAMMING

Standard C++ | Directives | CUDA

Coming soon to an NVIDIA HPC C++ compiler near you

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

```
#pragma acc data copyin(x) copy(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++

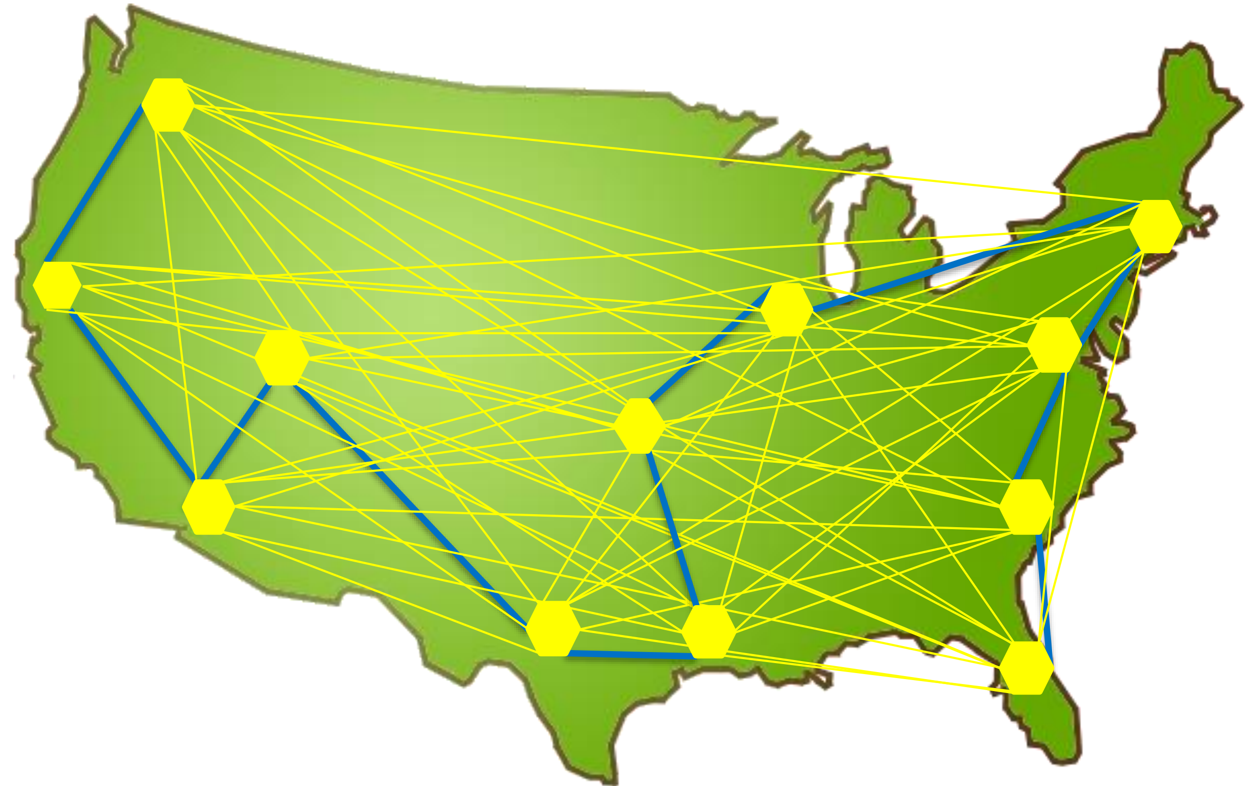
Incremental Performance Optimization with OpenACC

Maximize GPU Performance with CUDA C++

EXAMPLES

TRAVELING SALESMAN

Find the shortest route that visits every city



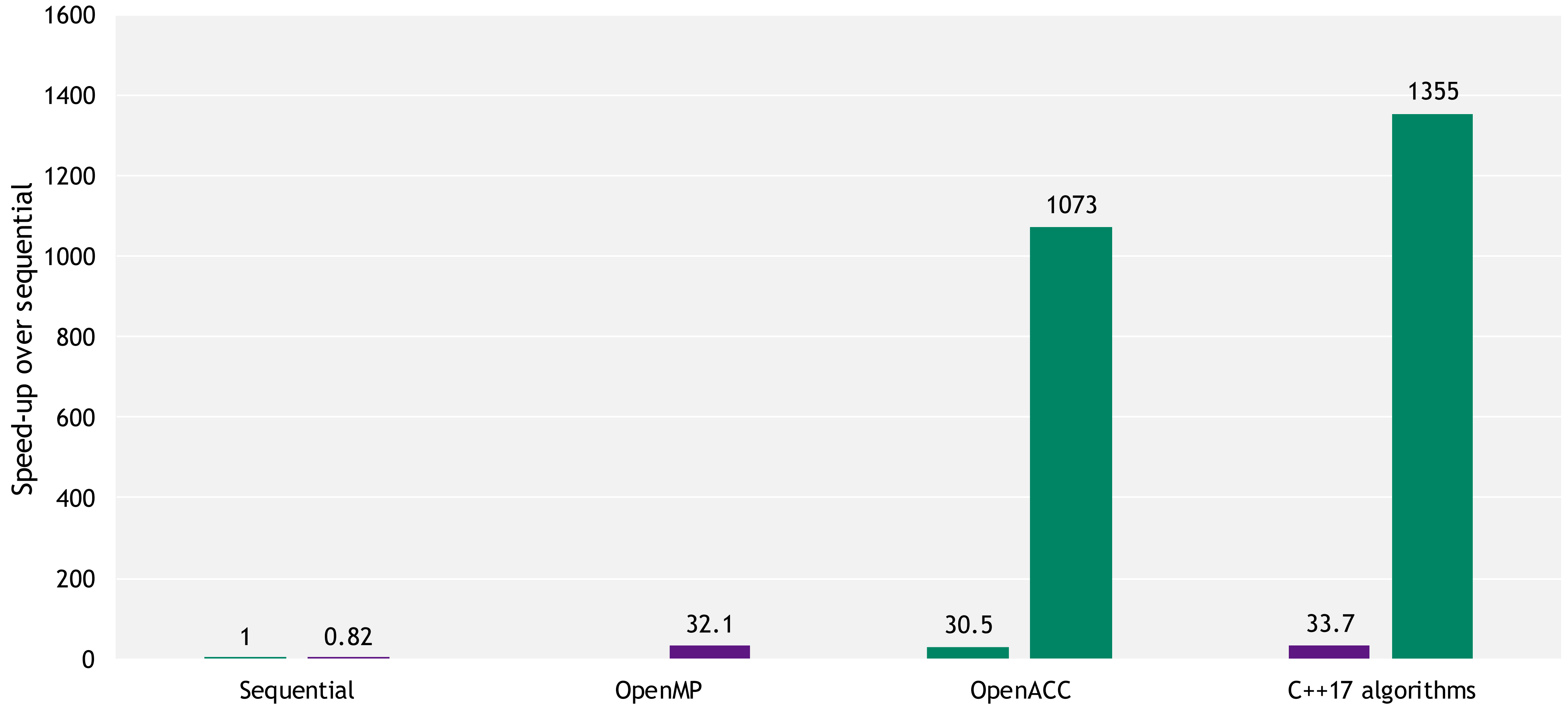
NVIDIA.

TRAVELING SALESMAN

```
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);
      });
}
```


NVIDIA

TRAVELING SALESMAN PERFORMANCE



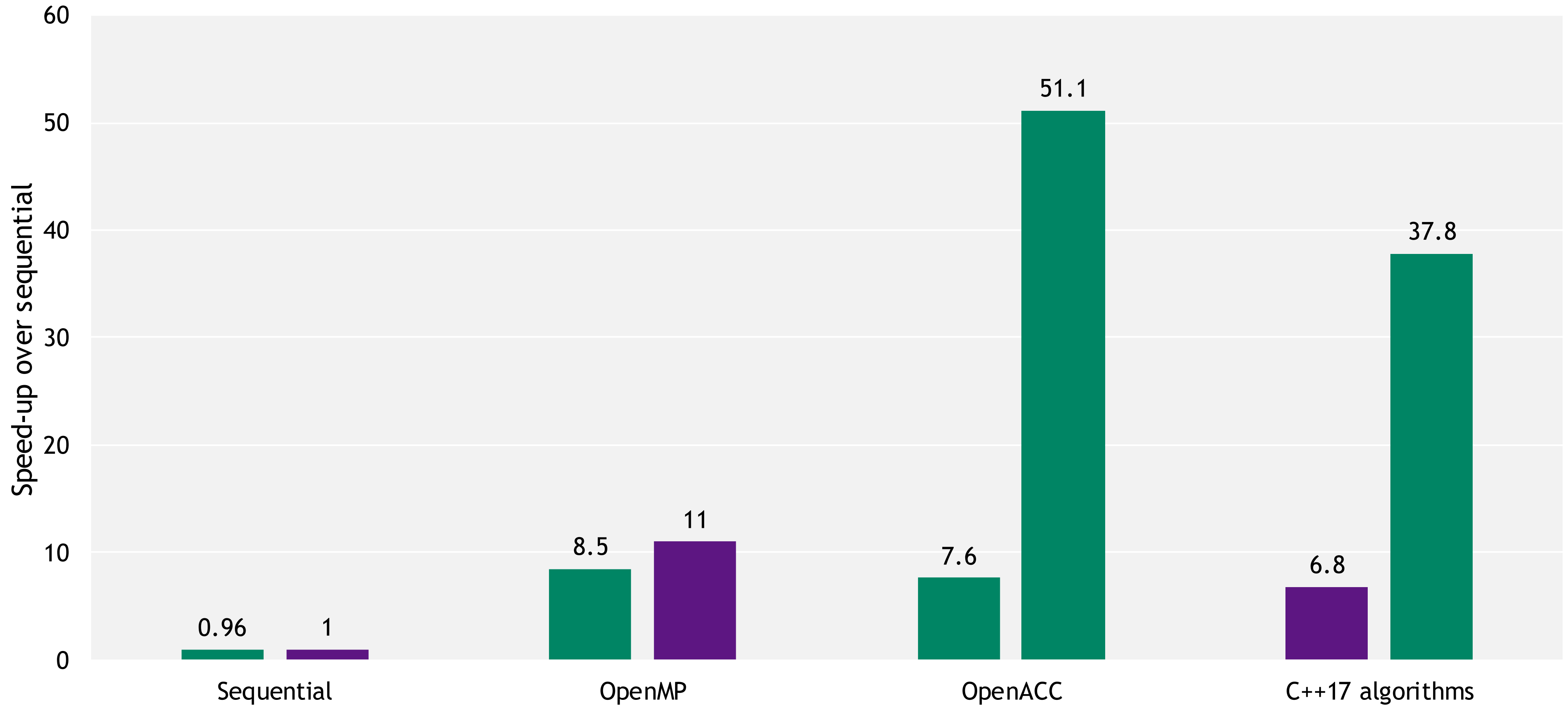
NVIDIA

JACOBI LAPLACE

```
int jacobi_solver(float* data, int M, int N,
                  float max_diff) {
  auto temp = std::make_unique<float[]>(M * N);
  std::copy(std::execution::par,
            data, data + M*N, temp.get());
  int iterations = 0;
  bool keep_going;
  float* from = data;
  float* to = temp.get();
  do {
    ++iterations;
    std::for_each(std::execution::par,
      counting_iterator<int>(N+1),
      counting_iterator<int>(((M-1)*N)-1),
      [=](int i) {
        if (i % N != 0 && i % N != N-1) {
          to[i] = 0.25f * (from[i-N] + from[i+N] +
                           from[i-1] + from[i+1]);
        }
      });
    keep_going = std::any_of(std::execution::par,
      counting_iterator<int>(N+1),
      counting_iterator<int>(((M-1)*N)-1),
      [=](int i) {
        return std::fabs(to[i] - from[i]) > max_diff;
      });
    std::swap(from, to);
  } while (keep_going);
  if (to == data) {
    std::copy(std::execution::par,
              temp.get(), temp.get() + M*N, data);
  }
  return iterations;
}
```


NVIDIA.

JACOBI LAPLACE PERFORMANCE



NVIDIA

GPU / CPU DIFFERENCES

AUTOMATIC DEVICE FUNCTIONS

No host/device annotations necessary

```
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);
    });
```

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
```


NVIDIA.

FUNCTION POINTERS

Use function objects or lambdas instead

```
void square(int& x) { x = x * x; }
struct x_squared {
        void operator()(int& x) const { x = x * x; }
};

// ...

std::for_each(std::execution::par, v.begin(), v.end(),
        x_squared()); // OK, function object

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

std::for_each(std::execution::par, v.begin(), v.end(),
        [](int& x) { square(x); }); // OK, lambda wrapper for regular function
```

MEMORY ISSUES

Unified Memory

PGI C++ parallel algorithms implementation uses unified memory

No data directives or function calls to move data

Heap memory is automatically shared

Stack memory and global memory are not shared

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 the heap

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


NVIDIA.

C++ STANDARD LIBRARY

C++ standard library not designed for GPUs

Not usable from CUDA device functions

NVIDIA

C++ STANDARD LIBRARY

CUDA

C++ standard library not designed for GPUs

~~Not usable from CUDA device functions~~

libcu++: opt-in, heterogeneous, incremental C++ standard library for CUDA

First release in CUDA 10.2

The CUDA C++ Standard Library, Bryce Adelstein Lelbach, Thursday, 2:00pm

NVIDIA

C++ STANDARD LIBRARY

Parallel algorithms

Many parts of the standard library are usable within parallel algorithm code

- Template functions
- Don’t throw exceptions
- No operating system dependency

Math functions (`sin`, `cos`, etc.)
`std::atomic<int>` and `std::atomic<long>`
`std::complex`
`std::optional`
`std::tuple`
`std::type_traits`
`std::pair`

NVIDIA

EXAMPLES, PART II

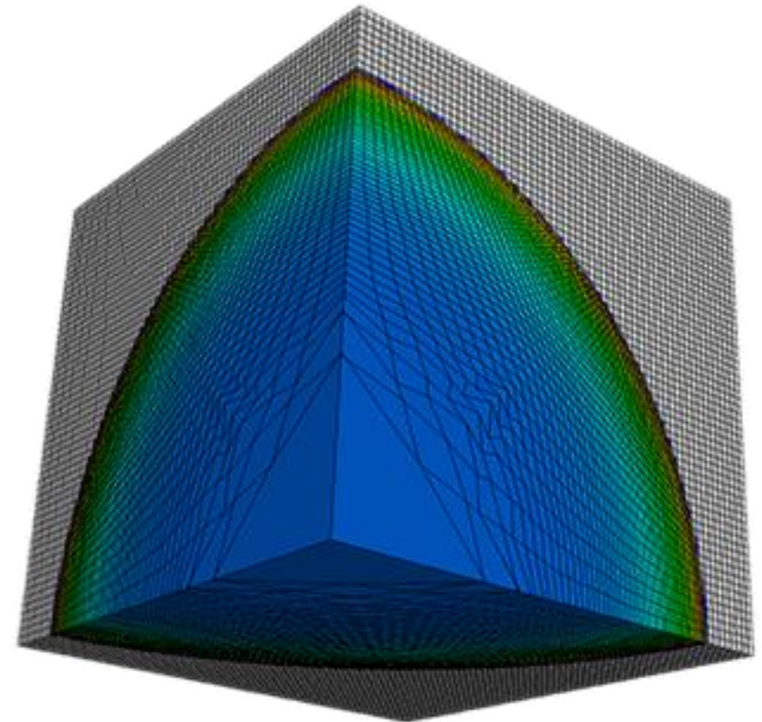
THE LULESH HYDRODYNAMICS MINI-APP

LLNL Hydrodynamics Proxy (Mini-App)

~9000 lines of C++

Parallel versions in MPI, OpenMP, OpenACC, CUDA, RAJA, Kokkos

Designed to stress compiler vectorization, parallel overheads, on-node parallelism



codesign.llnl.gov/lulesh

NVIDIA

LULESH

```
#pragma omp parallel for \
            firstprivate(numElem, hourg)
  for(Index_t i2=0;i2<numElem;++i2){
```

```
std::for_each_n(std::execution::par,
    counting_iterator(0), numElem,
    [=, &domain](Index_t i2) {
```

```
#pragma omp parallel for firstprivate(length)
  for (Index_t i = 0; i < length ; ++i) {
    Real_t c1s = Real_t(2.0)/Real_t(3.0) ;
    bvc[i] =
         c1s * (compression[i] + Real_t(1.));
    pbvc[i] = c1s;
  }
```

```
constexpr Real_t cls =
                 Real_t(2.0) / Real_t(3.0);
std::transform(std::execution::par,
    compression, compression + length, bvc,
    [=](Real_t compression_i) {
      return cls * (compression_i +
                    Real_t(1.0));
    });
std::fill(std::execution::par,
          pbvc, pbvc + length, cls);
```

C++ with OpenMP

Parallel C++17

NVIDIA.

```
static inline
void CalcHydroConstraintForElems(Domain &domain, Index_t length,
                                  Index_t *regElemlist, Real_t dvovmax, Real_t& dthydro)
{
#if _OPENMP
   const Index_t threads = omp_get_max_threads();
   Index_t hydro_elem_per_thread[threads];
   Real_t dthydro_per_thread[threads];
#else
   Index_t threads = 1;
   Index_t hydro_elem_per_thread[1];
   Real_t  dthydro_per_thread[1];
#endif
#pragma omp parallel firstprivate(length, dvovmax)
   {
      Real_t dthydro_tmp = dthydro ;
      Index_t hydro_elem = -1 ;
#if _OPENMP
      Index_t thread_num = omp_get_thread_num();
#else
      Index_t thread_num = 0;
#endif
#pragma omp for
      for (Index_t i = 0 ; i < length ; ++i) {
         Index_t indx = regElemlist[i] ;

         if (domain.vdov(indx) != Real_t(0.)) {
            Real_t dtdvov = dvovmax / (FABS(domain.vdov(indx))+Real_t(1.e-20)) ;

            if ( dthydro_tmp > dtdvov ) {
                  dthydro_tmp = dtdvov ;
                  hydro_elem = indx ;
            }
         }
      }
      dthydro_per_thread[thread_num]    = dthydro_tmp ;
      hydro_elem_per_thread[thread_num] = hydro_elem ;
   }
   for (Index_t i = 1; i < threads; ++i) {
      if(dthydro_per_thread[i] < dthydro_per_thread[0]) {
         dthydro_per_thread[0]    = dthydro_per_thread[i];
         hydro_elem_per_thread[0] =  hydro_elem_per_thread[i];
      }
   }
   if (hydro_elem_per_thread[0] != -1) {
      dthydro =  dthydro_per_thread[0] ;
   }
   return ;
}
```

C++ with OpenMP

LULESH

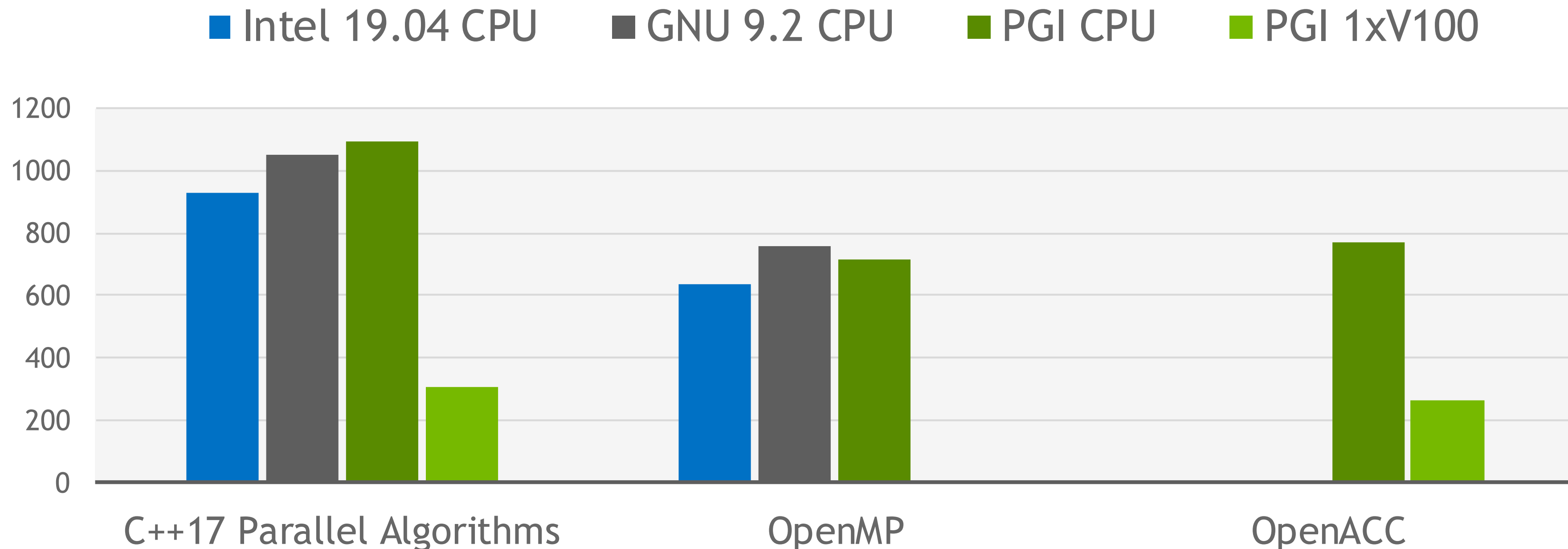
std::transform_reduce handles of the boilerplate code for the reduction

```
static inline void CalcHydroConstraintForElems(
    Domain &domain, Index_t length,
    Index_t *regElemlist, Real_t dvovmax,
    Real_t &dthydro)
{
  dthydro = std::transform_reduce(std::execution::par,
      counting_iterator(0), counting_iterator(length),
      dthydro,
      [](Real_t a, Real_t b) { return a < b ? a : b; },
      [=, &domain](Index_t i) {
        Index_t indx = regElemlist[i];
        if (domain.vdov(indx) == Real_t(0.0)) {
          return std::numeric_limits<Real_t>::max();
        } else {
          return dvovmax /
                 (std::abs(domain.vdov(indx)) +
                  Real_t(1.e-20));
        }
      });
}
```

Parallel C++17

LULESH 150^3 PERFORMANCE

Time in Seconds – Smaller is Better



Performance measured November 2019.
CPU: Two 20 core Intel Xeon Gold 6148 CPUs @ 2.4GHz w/376GB memory, hyperthreading enabled.
GPU: NVIDIA Tesla V100-PCIe-16GB GPU @ 1.53GHz.

CONCLUSION

THE FUTURE OF GPU C++ PROGRAMMING

Standard C++ | Directives | CUDA

Coming soon to an NVIDIA HPC C++ compiler near you

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

```
#pragma acc data copyin(x) copy(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++

David Olsen | PGI C++ compiler | https://www.pgicompilers.com

NVIDIA

NVIDIA