Performance analysis of a stencil code in modern C++

Victor Eijkhout, Yojan Chitkara, Daksh Chaplot

I. Introduction

In this paper we evaluate multiple parallel programming models with respect to both ease of expression, and resulting performance. We do this by implementing the mathematical algorithm known as the 'power method' in a variety of ways, using modern C++ technniques.

II. Algorithm aspects

We briefly discuss the power method, mostly going into its computational aspects. For the mathematical side, see HPC book, section **??**.

Computationally, the power method is an attractive paradigmatic example in that it exhibits the most common parallelism patterns:

- independent or 'convenient' parallelism;
- global reduction operations
- point-to-point communications.

As follows. The method is given by

Let A a matrix of interest
Let x be a random vector
For iterations until convergence
    compute the product $y \leftarrow Ax$
    compute the norm $\gamma = \|y\|$
    normalize $x \leftarrow y/\gamma$

where in the limit γ will be the largest eigenvalue of A, and x the corresponding eigenvector.

eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin.

The basis to parallelizing the power method lies in the distribution of x. Each processing element p, whether that be a core or processor, takes care (in a sense that we will later defined) of a disjoint set of indices I_p. The operations then are:

- The normalization step has independent parallelism: each component $y_i = x_i/\gamma$ can be independently computed, so each process p can compute y_i for $i \in I_p$ fully independently;
- Computing γ is a all-reduction: each processing element p first computes $\gamma_p = \sqrt{\sum_{i \in I_p} y_i^2}$ independently, and $\gamma = \sqrt{\sum_p \gamma_p}$ is then formed by combination, and made available to all p;
- Finally, for the operator A we assume a sparse matrix, for instance a Finite Difference (FD) stencil for a Partial Diffential Equation (PDE), or a *convolution* kernel. This is characterized by each component y_i of the output needing several components x_j of the input. In our case, putting a two-dimensional indexing on the vectors:

$$y_{ij} = 4x_{ij} - x_{i-1,j} - x_{i+1,j} - x_{i,j-1} - x_{i,j+1}$$

A. Loop parallelism

In our power method application a first example of a parallel loop is scaling an array by a factor.

```
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ )
    out[ IINDEX(i,j,m,n,b) ] = in[
     IINDEX(i,j,m,n,b) ] * factor;
```

where m,n are the size of the domain and b is the width of a halo region.

Some remarks:

1) We use template parameters `idxint` and `real` for integers and floating point quantities, respectively.
2) The loops range only over the $m \times n$ interior part of the domain, which is allocated with size $(m+2b) \times (n+2b)$.
3) We are using a traditional macro to translate from two-dimensional to one-dimensional indexing, skipping the border points:

   ```
   // seq.cpp
   #define IINDEX( i,j,m,n,b ) ((i)+b)
       *(n+2*b) + (j)+b
   ```

   Later we will discuss other indexing schemes.
4) For the initial implementation, the `out`, `in` data are `double*` pointers, for instance obtained as the `data` member of a `std::vector`. Later we will explore implementing the data as an `mdspan` object so we can use the two-dimensional `[i,j]` type indexing.

B. Reduction

The ℓ_2 reduction of our example application looks in code like:

```
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ ) {
    auto v = out[ IINDEX(i,j,m,n,b) ];
    sum_of_squares += v*v;
  }
return std::sqrt(sum_of_squares);
```

OpenMP resolves the race condition on the reduction variable by giving each thread a local copy of `sum_of_squares`, summing into that, and adding the local copies together in the end. This is illustrated in figure 1.

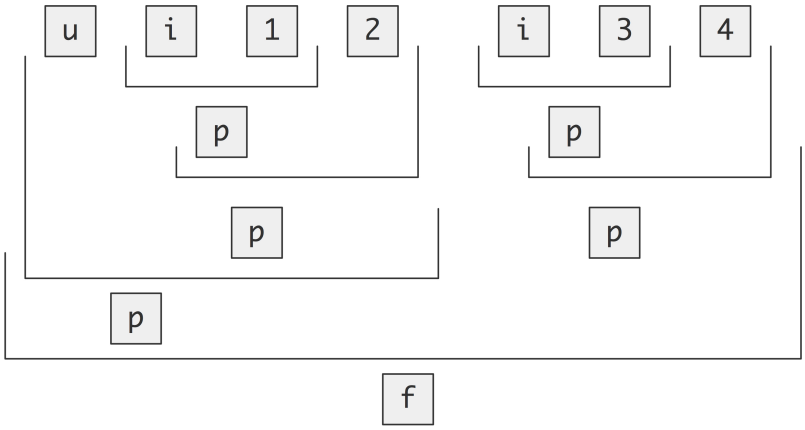

Fig. 1. Structure of a reduction of four items on two threads, where u is the user-supplied initial value, and i is the natural initial value for the reduction operator.

C. Five-point stencil

The evaluation of the 5-point difference operator is perfectly parallel in that there are no dependencies between the points of the output vector. However, the loop is more complicated than the scaling operation above:

```
for ( idxint i=0; i<m; i++ ) {
  for ( idxint j=0; j<n; j++ ) {
    out[ IINDEX(i,j,m,n,b) ] = 4*in[
     IINDEX(i,j,m,n,b) ]
      - in[ IINDEX(i-1,j,m,n,b) ] - in[
     IINDEX(i+1,j,m,n,b) ]
      - in[ IINDEX(i,j-1,m,n,b) ] - in[
     IINDEX(i,j+1,m,n,b) ];
  }
}
```

While this operation is somewhat like a 'transform' range operation, the complexity of the right-hand-side indexing precludes using an actual range algorithm. Therefore, in later versions of this code we will range over the indices, rather than the actual data.

III. Parallelization strategies

There are various ways we can parallelize our example application.

A. *OpenMP loop parallelism*

Above in section II-A we already showed the simplest parallelization scheme: we loop two-dimensionally over the index space, and translate that through a C Preprocessor (CPP) macro to one-dimensional indexing in a traditional container. The loops are then parallelized by OpenMP.

In graphs to follow we indicate by ‘oned’ the mode where we only parallelize the outer loop. For instance, the norm calculation is:

```
#pragma omp parallel for reduction
  (+:sum_of_squares)
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ ) {
    auto v = out[ INDEX(i,j,m,n,b) ];
    sum_of_squares += v*v;
  }
return std::sqrt(sum_of_squares);
};
```

We designate by ‘clps’ this same code, but adding `collapse(2)` to each OpenMP loop nest.

B. `mdspan` *indexing*

While the scaling and the norm calculation can be expressed as range algorithms, doing so for the five-point stencil update is trickier, so we take a two-step approach:

1) We access data through an `mdspan`, so that have multidimensional indexing; see figure 2
2) We define the iteration space as a `cartesian_product` range view:

```
const auto& s = data2d();
int b = this->border();
idxint
  lo_m = static_cast<idxint>(b),
  hi_m = static_cast<idxint>(s.
    extent(0)-b),
  lo_n = static_cast<idxint>(b),
  hi_n = static_cast<idxint>(s.
    extent(1)-b);
range2d = rng::views::
    cartesian_product
  ( rng::views::iota(lo_m,hi_m),
    rng::views::iota(lo_n,hi_n) )
    ;
```

This allows us to write, cleanly and compactly:

```
// span.cpp
auto out = this->data2d();
const auto in = other.data2d();
#pragma omp parallel for
for ( auto ij : this->inner() ) {
  auto [i,j] = ij;
  out[ i,j ] = 4*in[ i,j ]
    - in[ i-1,j ] - in[ i+1,j ] - in[ i,j
     -1 ] - in[ i,j+1 ];
}
```

Unlike the OpenMP double loop, we now have a single loop spanning the two-dimensional domain. Also, note that this domain is the interior of a larger domain.

In graphs to follow we indicate this mode by ‘span’.

In the tests below it will become apparent that the two-dimensional iteration space is not without performance problems. We tackle this by having a variant, ‘iota’, which splits the space orthogonally:

```
#pragma omp parallel for reduction
  (+:sum_of_squares)
for ( auto i : this->inneri() )
  for ( auto j : this->innerj() ) {
      auto v = array[i,j];
      sum_of_squares += v*v;
  }
return std::sqrt(sum_of_squares);
};
```

Class data:

```
private:
  real *_data{nullptr};
  md::mdspan<
    real,
    md::dextents<idxint,2>
          > cartesian_data;
```

Accessor:

```
//! pointer to the data as 2D array
auto& data2d() {
  return cartesian_data; };
const auto& data2d() const {
  return cartesian_data; };
```

Fig. 2. Use of `mdspan` for 2D data access

C. DIY cartesian product

The `cartesian_product` view is quite general so one could wonder about overhead. We write a custom iterator over a contiguous domain. It maintains internally `i,j` coordinates which are updated by the `operator++`. A naive implementation uses integer division, so we also make an optimized implementation without division or branching.

Simple

```
auto& operator++(
    ) {
  j++; i+= (j/m); j
    = j%m;
  return *this; };
```

Optimized

```
auto& operator++(
    ) {
  c--;
  j++; j *= (c>0);
  i += (c==0);
  c += m*(c==0);
  return *this;
};
```

D. Kokkos and Sycl

We also use the Kokkos and Sycl libraries in their 'host' mode. In graphs to follow we indicate these modes by '`kokkos`' and '`sycl`' respectively.

1) Sycl: Sycl is an open standard that targets heterogeneous parallelism through strict standard C++. It has mechanisms for memory management between host CPU and devices (both GPU and FPGA), and for expressing common parallel algorithms.

The preferred mechanism for handling memory coherence is through buffers. Host memory is wrapped in a buffer structure:

```
std::vector<real> Mat_A(msize*nsize);
buffer<real,2> Buf_a(Mat_A.data(),
    range<2>(msize,nsize));
```

This memory can then transparently be accessed in a kernel, whether run on host or device, where coherence is ensured by the runtime. This turns out to be as efficient as more explicit mechanisms.

```
q.submit([&] (handler &h) {
  accessor D_a(Buf_a,h,write_only);
  h.parallel_for
    (range<2>(msize-2,nsize-2),
     [=](auto index){
       auto row = index.get_id(0) + 1;
       auto col = index.get_id(1) + 1;
       D_a[row][col] = 1.;
     });
}).wait();
```

Note that Sycl has a true two-dimensional indexing mode.

We see that the `parallel_for` construct resembles a C++ range algorithm: it combines a range – though of the index space, not the data space – and a lambda expression to be applied at each point.

Sycl is alone among the models studied here in that it requires index shifting in order to range over the interior of the index space.

2) Kokkos: Kokkos is the execution layer of the Trilinos project. It is, like Sycl, a data-parallel programming mode that supports host and device execution with the same code. Unlike Sycl there is no explicit queue; instead, objects are explicitly associated with the host or device data space.

```
// diff2d.cpp
using MemSpace = Kokkos::HostSpace;
using Layout = Kokkos::LayoutRight;
using HostMatrixType = Kokkos::View<
    real**, Layout, MemSpace>;
HostMatrixType x("x", msize,nsize );
```

Like Sycl, there are explicit parallelism constructs. These closely resemble C++ range algorithms, specifying an explicit index space over which to iterate, and the lambda expression to apply at each index.

```
Kokkos::parallel_for
  ("Update x",
   Kokkos::MDRangePolicy<Kokkos::Rank
    <2>>
       ({1, 1}, {msize-1, nsize-1}),
   KOKKOS_LAMBDA(int i, int j) {
    x(i, j) = Ax(i, j) / norm;
  });
```

IV. Timing comparison and discussion

We perform different comparisons:

- different parallelization models on a given compiler and CPU;
- different Intel processor generations, given a model and compiler;
- different compilers, given a model and CPU.

A. Comparing parallelization models

Let's compare various implementation strategies. Test given here are on an *Intel Sapphire Rapids* dual socket node with 112 cores total. We compare the Intel 2024 (figure 3) and GCC 13 (figure 4) compilers.

We make the following observations.

1) Intel compiler: First of all, we see that the naive '`oned`' OpenMP implementation is clearly the fastest. All of the schemes that use two-dimensional iteration are slower. An indication of the reason for this is outlined in section IV-D: much of the computing time actually goes into incrementing the two-dimensional iterator.

If we accept a constant loss of efficiency for these schemes, we see that most schemes even have a descending efficiency with increasing core count. The reason for this is not clear, especially since the '`clps`' and '`diy`' implementations do not show this.

Interestingly, on low core counts the '`sycl`' and '`kokkos`' implementations are relatively competitive. They probably employ an efficient translation to an OpenMP backend, but this makes their gradual loss of efficiency all the more puzzling.

2) GNU compiler: For the GNU compiler the differences between the various models are much smaller. Moreover, most models have constant efficiency, kokkos being the exception.

B. Comparing processor generations

In figure 5 we compare the following generations of Intel processors:

- *Intel Sky Lake* in the *TACC Stampede2* cluster, in a two-socket configuration with a total of 48 cores;
- *Intel Cascade Lake* in the *TACC Frontera* cluster, in a two-socket configuration with a total of 56 cores;
- *Intel Ice Lake* in the *TACC Stampede3* cluster, in a two-socket configuration with a total of 80 cores;
- *Intel Sapphire Rapids* in the *TACC Stampede3* cluster, in a two-socket configuration with a total of 112 cores;

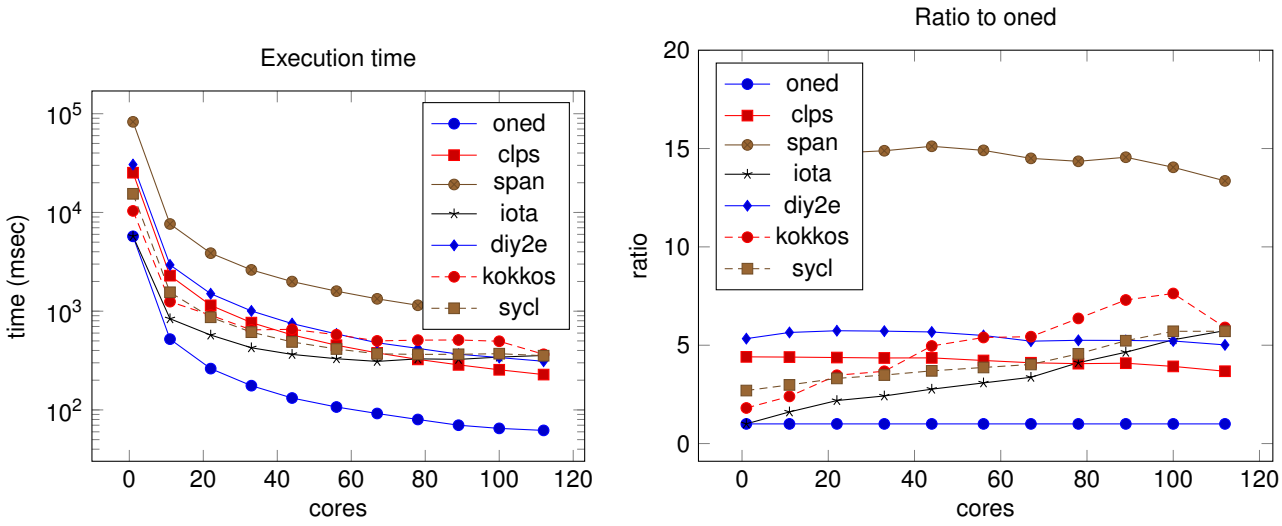

Fig. 3. Comparing implementation strategies, Intel 2024 compiler on a 112-core Sapphire Rapids node.

- *Intel Granite Rapids* in the *TACC Stampede3* cluster, in a two-socket configuration with a total of 240 cores;

1) Runtime: In the left graph we see that the runtimes are roughly equal for low core counts; the main difference is that Sky Lake and Cascade Lake reach their maximum performance well short of the total core count, while Ice Lake does not show this behavior. The Ice Lake and later generations keep scaling with increasing core count, the both Rapids generations even almost linearly.

2) Bandwidth: Our bandwidth measure is not observed directly, but rather derived by considering the amount of data loads in the algorithm:

- The five-point stencil application loads 5 elements from the input, loads the output vector, and writes it; this would come to 7 data accesses per i,j point calculation.
- However, subsequent points from each i or $i-1$ or $i+1$ line come from the same cacheline, so effectively we 3 accesses from the input vector, for 5 total.
- Added to this, lines easily fit in L2 cache, so after the line for one $i+1$ value has been loaded, it will be used as the i line in the next iteration, and the $i-1$ line in the iteration thereafter. Thus, we really have only 1 DRAM access for the input vector per i,j point calculation, for a total of 3 access.
- Finally, the Ice Lake processor can, in certain circumstances, convert the calculation to a 'streaming store', so the load of the output vector doesn't need to be counted.

The right graph of figure 5 shows a *a posteriori* measure for attained bandwidth.

We see that the Sky Lake and Cascade Lake processors bear out the old truism that the total bandwidth is less than each core's individual bandwidth summed over the cores.

However, newer processor generations scale further, and for the Sapphire Rapids and Granite Rapids even fairly linear. The SPR node has Intel's 'High Bandwidth Memory', given a

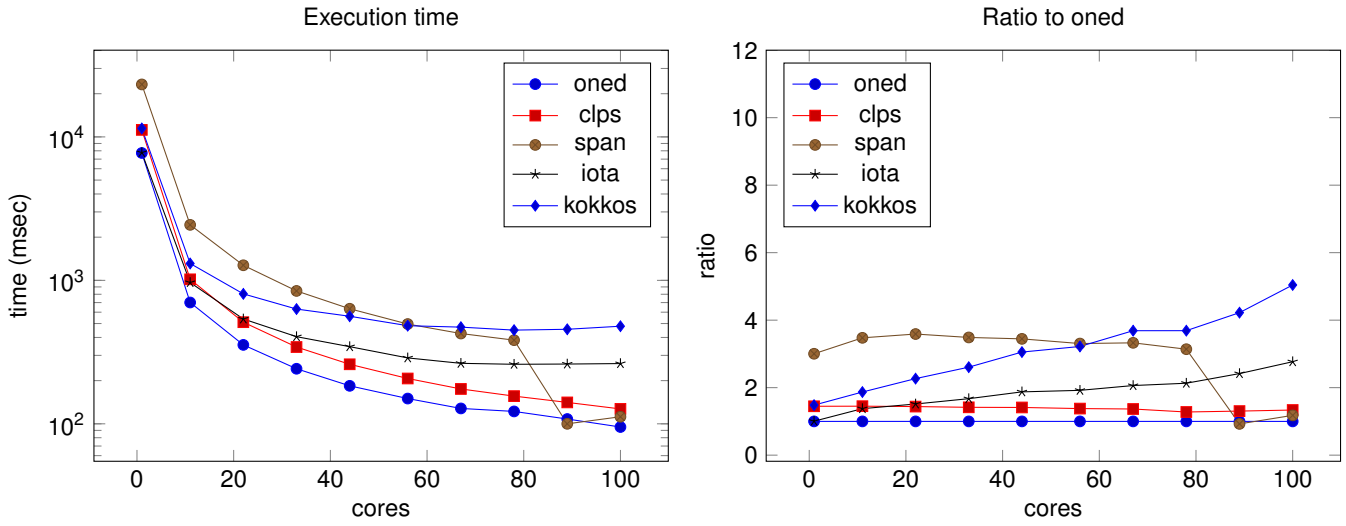

Fig. 4. Comparing implementation strategies, Gcc 2024 compiler on a 112-core Sapphire Rapids node.

much higher absolute number.

C. Comparing compilers

Figures 6 and 7 show a comparison between the Intel 2024 and GCC 13 compilers. While sometimes Intel has a slight edge over GCC, it sometimes performs considerably worse. Inspection of the generated code would be needed to determine the cause of this.

D. Analysis

One immediate conclusion from the above tests is that it's hard to beat the simple-minded OpenMP implementation with only the outer loop parallelized, although differences are less pronounced for the Gnu compiler than the Intel one.

Doing profiling gives us a clue. Here is the output of running the `mdspan` / `cartesian_product` version (Intel 24 compiler):

```
%% make run_perf VARIANTS=span NSIZE
    =10000 ECHO=1

    55.60%  [.] std::ranges::
    cartesian_product_view<std::
    ranges::iota_view<long, long>, std
    ::ranges::iota_view<long, long>
    >::_Iterator<true>::operator+=
    18.73%  [.] __divti3
    11.33%  [.] linalg::
    bordered_array_span<float>::
    central_difference_from
     5.37%  [.] linalg::
    bordered_array_span<float>::
    scale_interior
     5.01%  [.] linalg::
    bordered_array_span<float>::
    l2norm
     2.69%  [.] __divti3@plt
```

We see that more than half of the time goes into index calculations, and in particular integer division.

VTune profiling tells us something similar (with some post-processing):

```
std::ranges::cartesian_product_view<
    std::ranges::iota_view<long, long
    >, std::ranges::iota_view<long,
```

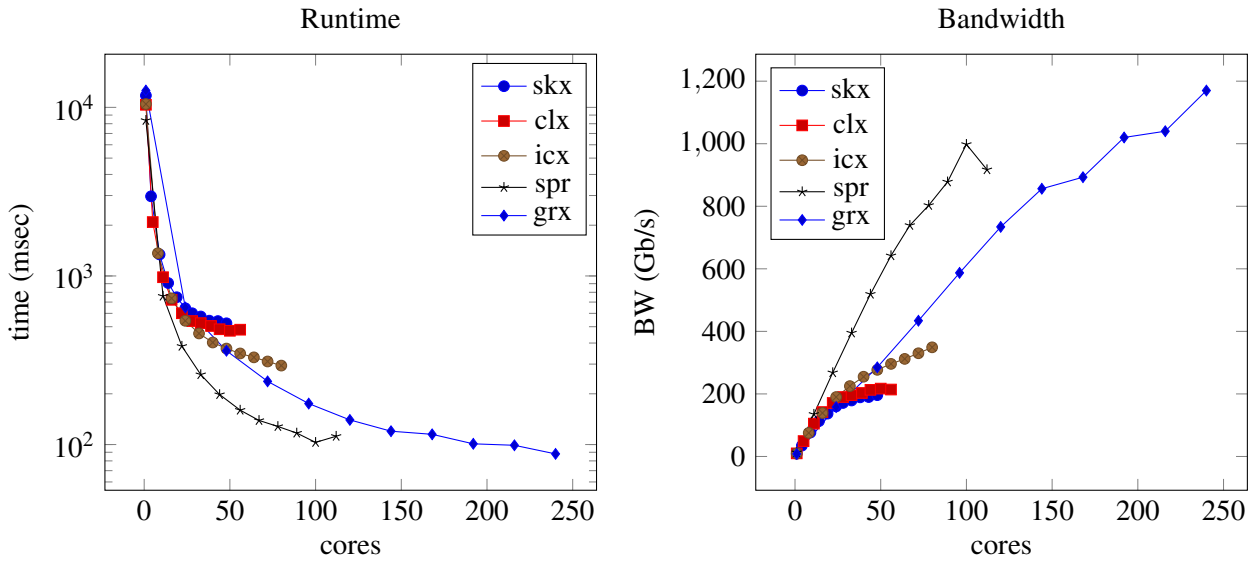

Fig. 5. Generations of Intel processors (OpenMP). Time (left) and speedup (right) as function of core count.

```
    long>>::_Iterator<(bool)1>::
    operator+= 44.15438
linalg::bordered_array_span<float>::
    central_difference_from 12.71539
                linalg::
    bordered_array_span<float>::
    l2norm 10.88307
         linalg::bordered_array_span<
    float>::scale_interior 9.63805

        func@0x404390 8.09852

            __divti3 8.09808

        __udivmodti4 4.61700
     linalg::bordered_array_span<float
    >::bordered_array_span 0.87979
                  linalg::
    bordered_array_span<float>::set
    0.87910

    __kmp_get_global_thread_id_reg
    0.00000
                    cfree 0.03664
```

V. Conclusion

We have implemented a stencil code, which stands for many scientific applications, using various modern C++ mechanisms. The tentative conclusion is that the modern mechanisms may be simpler to code, but this elegance comes with a certain perforance penalty, more so for the Intel compiler than the GCC compiler.

Acknowledgement: This work was supported by the Intel OneAPI Center of Excellence, and the TACC STAR Scholars program, funded by generous gifts from TACC industry partners, including Intel, Shell, Exxon, and Chevron.

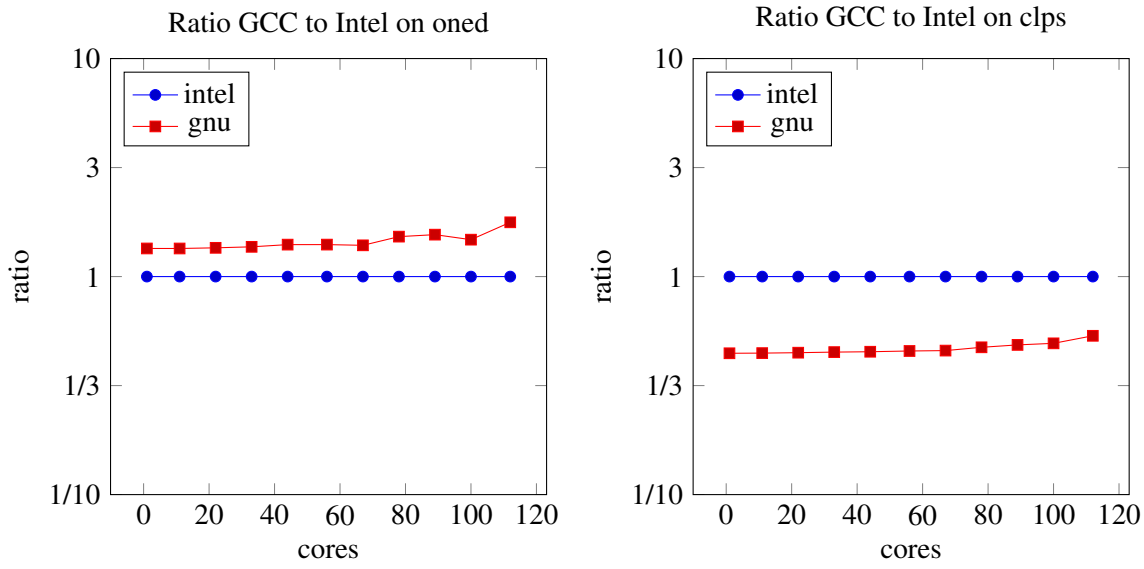

Fig. 6. Comparing Intel to GCC on 'oned' (left) and 'clps' (right) scheme

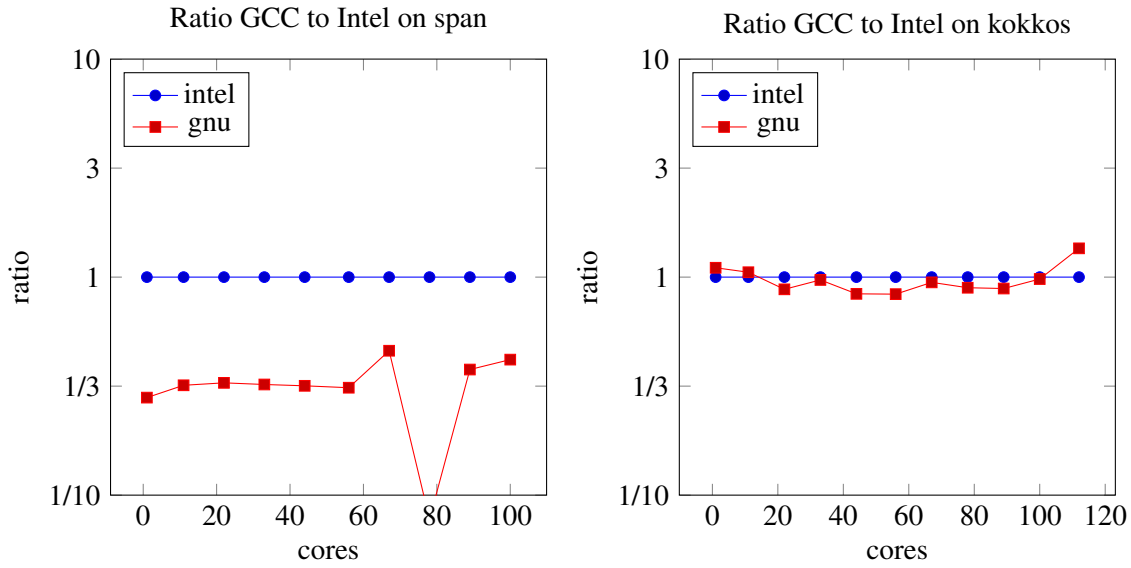

Fig. 7. Comparing Intel to GCC on 'span' (left) and 'kokkos' (right) scheme