Performance analysis of a stencil power method

Victor Eijkhout[*], Yojan Chitkara, Daksh Chaplot

August 9, 2024

Part I
Writeup

1 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, for the moment all in C++.

2 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 $x \leftarrow y/\gamma$

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

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

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:

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

2.1 Loop parallelisme

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,b,n2b) ] = in[ IINDEX(i,j,b,n2b) ] * 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 $(m+b) \times (n+b)$.
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,b,n2b ) ((i)+b)*n2b + (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.

2.2 Reduction

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

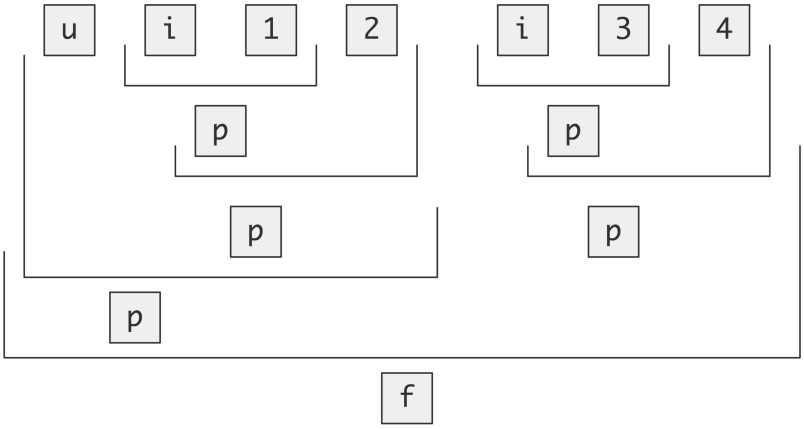

Figure 1: Structure of a reduction of four items on two threads, where u is the user-supplied initial value, and i is the natural initial value for the reduction operator.

```
for ( idxint i=0; i<m; i++ )
  for ( idxint j=0; j<n; j++ ) {
    auto v = out[ IINDEX(i,j,b,n2b) ];
    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.

2.3 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,b,n2b) ] = 4*in[ IINDEX(i,j,b,n2b) ]
      - in[ IINDEX(i-1,j,b,n2b) ] - in[ IINDEX(i+1,j,b,n2b) ]
      - in[ IINDEX(i,j-1,b,n2b) ] - in[ IINDEX(i,j+1,b,n2b) ];
  }
}
```

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.

3 Parallelization strategies

There are various ways we can parallelize our example application.

3.1 OpenMP loop parallelism

Above in section **??** 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:

```
template< typename real >
real bordered_array_1d<real>::l2norm() {
  real sum_of_squares{0};
  auto out = this->data();
#   pragma omp parallel for reduction(+:sum_of_squares)
  for ( size_t i=0; i<_m; i++ )
    for ( size_t j=0; j<_n; j++ ) {
      auto v = out[ IINDEX(i,j) ];
      sum_of_squares += v*v;
    }
  log_flops(_m*_n*3); log_bytes( sizeof(real)*_m*_n*1 );
  return std::sqrt(sum_of_squares);
};
```

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

3.2 `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:

   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; };
   ```

2. We define the iteration space as a `cartesian_product` range view:

   ```
   mutable std::optional< decltype( rng::views::cartesian_product
                         ( rng::views::iota(idxint{0},idxint{0}),
                           rng::views::iota(idxint{0},idxint{0}) ) ) >
         range2d = {};
   auto inner() const {
     if (not range2d.has_value()) {
       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) );
    }
    return *range2d;
  };
```

This allows us to write, cleanly and compactly:

```
// span.cpp
template< typename real >
void bordered_array_span<real>::central_difference_from
    ( const linalg::bordered_array_base<real>& _other,bool trace ) const {
  const auto& other =
    dynamic_cast<const linalg::bordered_array_span<real>&>(_other);
  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 ];
  }
};
```

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

3.3 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 as follows:

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;
      };
```

3.4 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.

3.4.1 Sycl

Sycl is an open standard that targets heterogeneous parallelism through strict standard C++.

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, 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.

Note the, somewhat contrived, index shifting that we have to apply in order to range over the interior of the index space.

3.4.2 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",
```

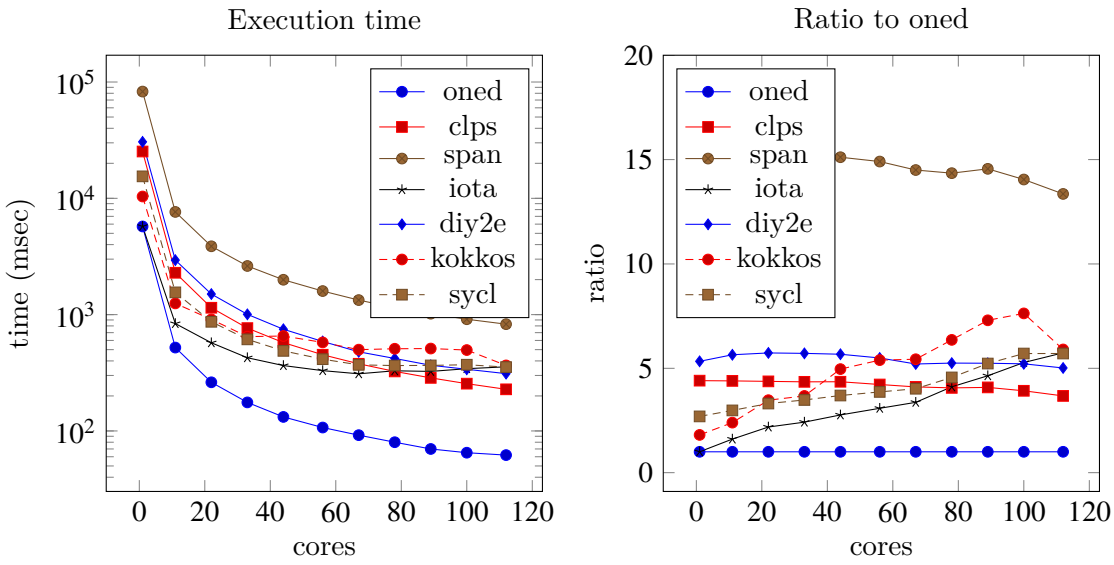

Figure 2: Comparing implementation strategies, Intel 2024 compiler on a 112-core Sapphire Rapids node.

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

4 Timing comparison and discussion

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 **??**) and GCC 13 (figure **??**) compilers.

In figure **??** we compare three generations of Intel processors:

- *Intel Sky Lake* in the *TACC Stampede2* cluster, in a two-socket configuration with a total of 40 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;

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.

In the right graph we read out the maximum bandwidth that is reached.

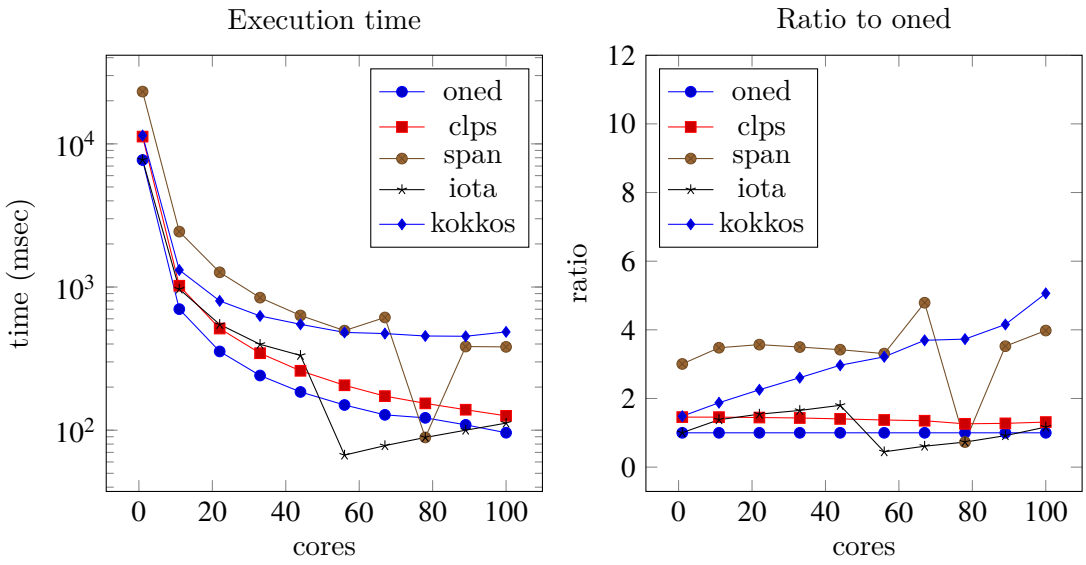

Figure 3: Comparing implementation strategies, Gcc 2024 compiler on a 112-core Sapphire Rapids node.

We need to start by explaining how we measure the bandwdith. We do this indirectly:

- 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 interesting figure here is the aggregate bandwidth. Usually this is less than the single-core bandwidth times the number of cores, but with the Ice Lake we see it scaling quite far.

Processor	single core bw attained/peak	aggregate bandwidth attained/peak
Cascade Lake	13/xx	230/281
Ice Lake	14/xx	307/409

4.1 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.

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

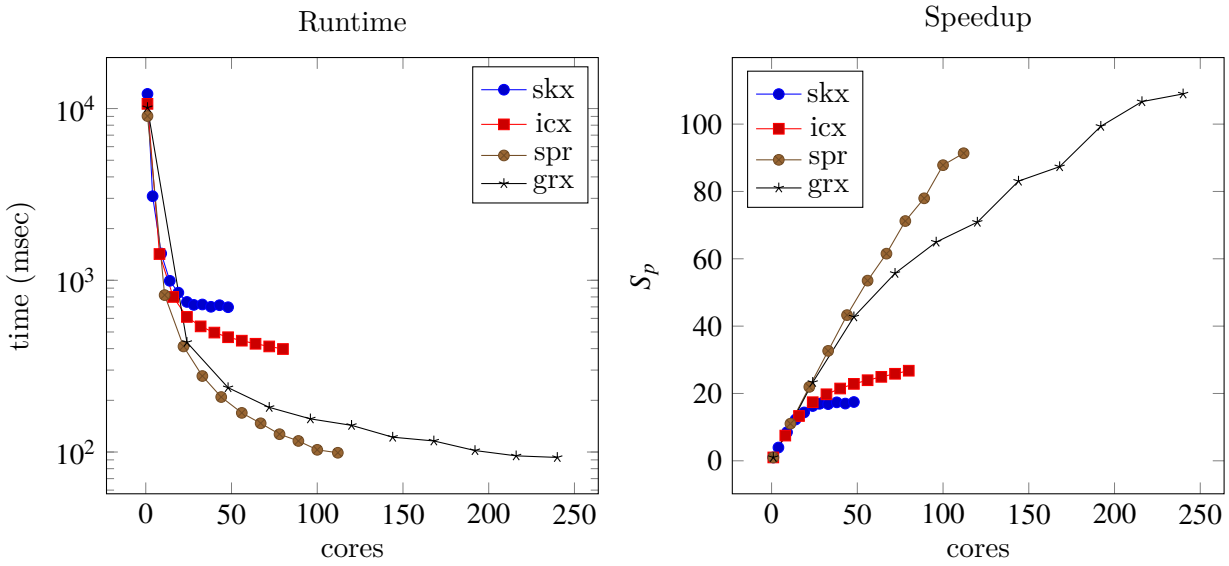

Figure 4: Generations of Intel processors (OpenMP)

```
%% 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. This makes send if we consider our 'diy' implementation:

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

Unfortunately it's not the plus-plus operator but the plus-and-is, which is the bottlenect. For the former we can come up with trickery to lose the divisions:

MISSING SNIPPET d2ddiziter in codesnippetsdir=../snippets

for the latter that's much harder. (Note that the tricked code has no conditionals that could give branch mispredictions!)

Unfortunately, *perf* does not help us much here:

```
    35.25%  [.] linalg::bordered_array_diy2e<float>::l2norm
    31.39%  [.] linalg::bordered_array_diy2e<float>::central_difference_from
    30.46%  [.] linalg::bordered_array_diy2e<float>::scale_interior
```

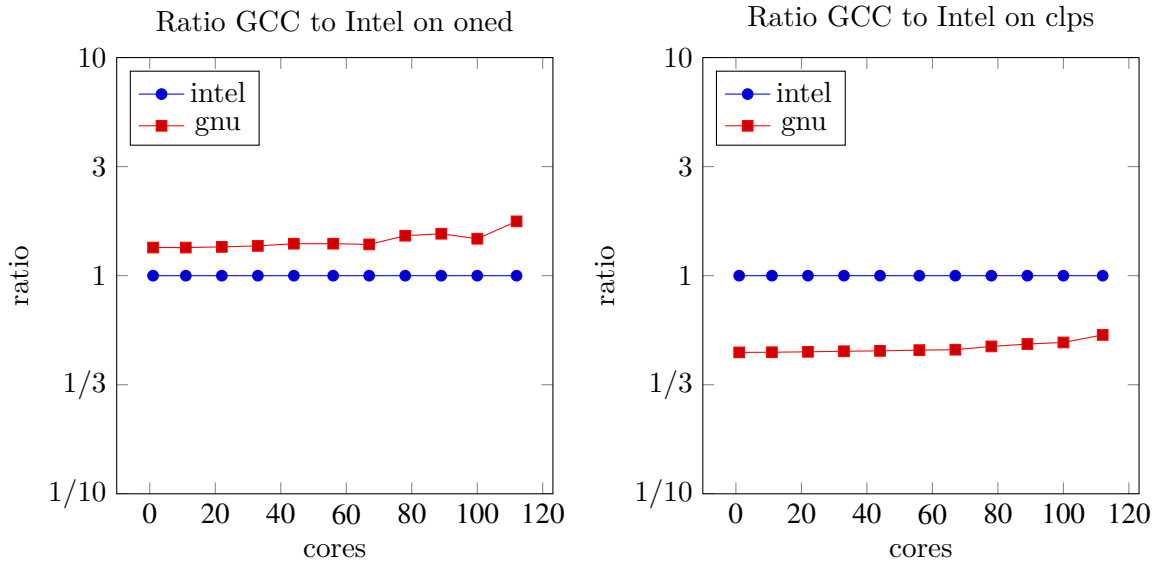

Figure 5: Comparing Intel to GCC on 'oned' (left) and 'clps' (right) scheme

```
2.29%  [.] linalg::bordered_array_diy2e<float>::set
```

We get no timings for the embedded iterator. Note the counterintuitive result that the norm computation takes more time than the central difference, despite the latter having more operations and more complicated memory access.

VTune profiling tells us the same as Perf (with some post-processing):

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

4.2 Comparing compilers

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.

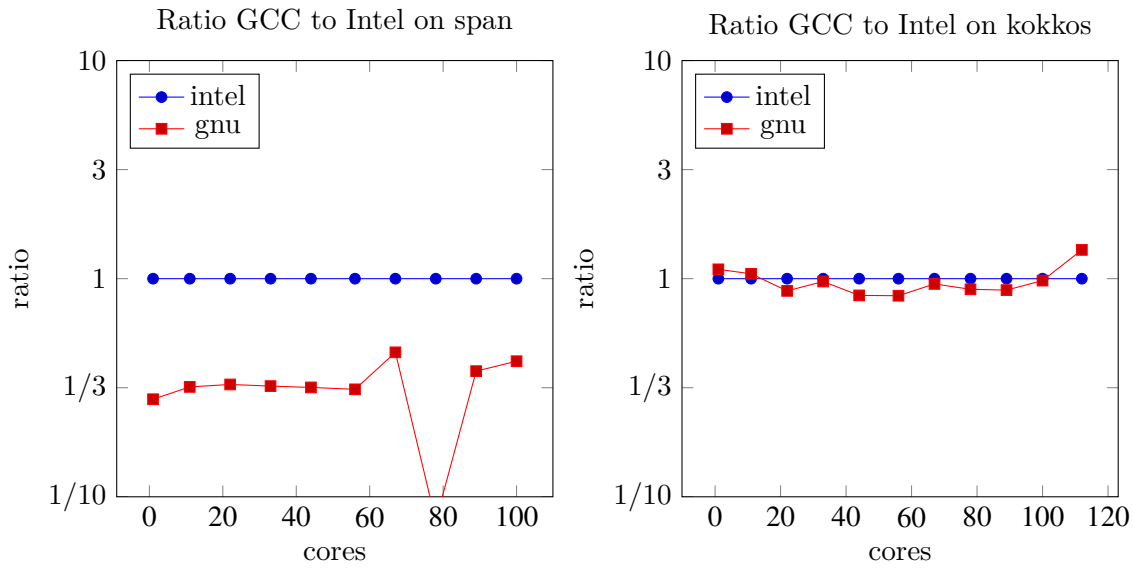

Figure 6: Comparing Intel to GCC on ‘span’ (left) and ‘kokkos’ (right) scheme

Part II

Appendix

5 Run scripts, settings, data

Compile:

```
## delete all binaries
make clean
## rebuild all
make cmake
```

You will get a listing of all binaries that have been built into the `bin` directory. Some versions may be missing for perfectly legitimate reasons, such as sycl for gcc.

The sofware relies on `cxxopts`, `mdspan`, `kokkos`. You can install those yourself, or let cmake download / install them for you.

Comparison of programming models:

```
./compare_models.sh -c spr -q spr-dev -p 112 -s -t -g all
```

where

- `-c spr` names the `spr` processor; this is purely for naming purposes.

This script is heavily SLURM-based.

- `-s` prepends `srun` with various parameters to the run command; the alternative is to run on a compute node.

- `-q spr-dev` names the SLURM queue if *`srun`* is used.
- `-p 112` gives the core count for the node if *`srun`* is used; otherwise *`SLURM`* parameters are queried.

The actual program run is done by another script that does a scaling study. As a result, a file with extension *`.out`* is generated.

- `-t` prints out trace information.
- `-g` indicates that the *`.out`* file 1. is renamed to *`.runout`*, and 2. is added to the repository. This of course requires you to have a writable fork.

6 Data

Table **??** left.

```
cores, oned, clps, span, iota, kokkos, sycl,
1, 5731, 25266, 82766, 5745, 10368, 15449,
11, 521, 2291, 7641, 838, 1249, 1554,
22, 262, 1146, 3870, 573, 910, 868,
33, 176, 766, 2620, 426, 647, 613,
44, 132, 575, 1995, 365, 655, 488,
56, 107, 452, 1595, 330, 577, 414,
67, 92, 378, 1334, 310, 500, 370,
78, 80, 325, 1148, 329, 509, 365,
89, 70, 286, 1019, 325, 511, 366,
100, 65, 255, 913, 343, 496, 371,
112, 62, 228, 828, 356, 366, 354,
```

Right.

```
cores, oned, clps, span, iota, kokkos, sycl,
1, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0,
11, 11.0, 11.028371890004365, 10.83182829472582, 6.855608591885441, 8.301040832666132,
    9.941441441441441,
22, 21.874045801526716, 22.047120418848166, 21.38656330749354, 10.026178010471204,
    11.393406593406594, 17.798387096774192,
33, 32.5625, 32.98433420365535, 31.590076335877864, 13.485915492957746,
    16.024729520865534, 25.202283849918434,
44, 43.416666666666664, 43.94086956521739, 41.48671679197995, 15.73972602739726,
    15.829007633587786, 31.6577868852459,
56, 53.5607476635514, 55.89823008849557, 51.89090909090909, 17.40909090909091,
    17.96880415944541, 37.31642512077295,
67, 62.29347826086956, 66.84126984126983, 62.04347826086956, 18.532258064516128, 20.736,
     41.75405405405405,
78, 71.6375, 77.74153846153847, 72.09581881533101, 17.462006079027354,
    20.36935166994106, 42.326027397260276,
89, 81.87142857142857, 88.34265734265735, 81.22276741903828, 17.676923076923078,
    20.289628180039138, 42.2103825136612,
100, 88.16923076923077, 99.08235294117647, 90.65279299014239, 16.749271137026238,
    20.903225806451612, 41.64150943396226,
112, 92.43548387096774, 110.8157894736842, 99.95893719806763, 16.1376404494382,
    28.327868852459016, 43.64124293785311,
```

Table **??** left.

```
cores, oned, clps, span, kokkos,
1, 7431, 10802, 23109, 11178,
11, 677, 983, 2333, 1187,
22, 343, 493, 1251, 762,
33, 232, 333, 816, 599,
44, 177, 250, 618, 537,
56, 146, 199, 486, 462,
67, 125, 169, 409, 441,
78, 121, 150, 375, 451,
89, 107, 135, 358, 466,
100, 96, 125, 372, 491,
112, 113, 120, 375, 517,
```

Right.

```
cores, oned, clps, span, kokkos,
1, 1.0, 1.0, 1.0, 1.0,
11, 10.976366322008863, 10.988809766022381, 9.905272181740248, 9.417017691659646,
22, 21.664723032069972, 21.91075050709939, 18.47242206235012, 14.669291338582678,
33, 32.0301724137931, 32.43843843843844, 28.31985294117647, 18.66110183639399,
44, 41.983050847457626, 43.208, 37.39320388349515, 20.81564245810056,
56, 50.897260273972606, 54.28140703517588, 47.54938271604938, 24.194805194805195,
67, 59.448, 63.917159763313606, 56.50122249388753, 25.346938775510203,
78, 61.413223140495866, 72.01333333333334, 61.624, 24.784922394678492,
89, 69.44859813084112, 80.01481481481481, 64.55027932960894, 23.987124463519315,
100, 77.40625, 86.416, 62.12096774193548, 22.765784114052952,
112, 65.76106194690266, 90.01666666666667, 61.624, 21.620889748549324,
```

Table **??** left.

skx

```
cores, skx,
1, 126585,
4, 31692,
9, 14099,
14, 9084,
19, 6699,
24, 5314,
28, 9078,
33, 8210,
38, 6704,
43, 5972,
48, 5658,
```

csx

```
cores, csx,
1, 109887,
5, 22172,
11, 10075,
16, 6934,
22, 5066,
```

```
28, 4092,
33, 3561,
39, 2888,
44, 2559,
50, 2304,
56, 7479,
```

icx

```
cores, icx,
1, 60636,
8, 9496,
16, 3840,
24, 2580,
32, 1949,
40, 1964,
48, 3249,
56, 2676,
64, 2341,
72, 1934,
80, 1917,
```

spr

```
cores, spr,
1, 52979,
11, 4852,
22, 2428,
33, 1626,
44, 1222,
56, 967,
67, 1597,
78, 1516,
89, 1211,
100, 1079,
112, 1164,
```