WPISA-CPR(18) 19527

G Latu et al.

Scaling and optimizing the Gysela code on a cluster of many-core processors

Preprint of Paper to be submitted for publication in Proceeding of Euro-Par 2018, 24th International European Conference on Parallel and Distributed Computing

This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

Scaling and optimizing the Gysela code on a cluster of many-core processors

Guillaume Latu[1], Yuuichi Asahi[1], Julien Bigot[2]
Tamas Feher[3], and Virginie Grandgirard[1]

[1] CEA, IRFM, 13108, Saint-Paul-lez-Durance, France
[2] Maison de la Simulation, CEA, CNRS, Univ. Paris-Sud, UVSQ,
Université Paris-Saclay, 91191 Gif-sur-Yvette, France
[3] Max Planck Institute for Plasma Physics, Boltzmannstr. 2, 85748 Garching, Germany

Abstract. The current generation of the Xeon Phi Knights Landing (KNL) processor provides a highly multi-threaded environment on which regular programming models such as MPI/OpenMP can be used. This specific hardware offers both large memory bandwidth and large computing resources and is currently available on computing facilities. Many factors impact the performance achieved by applications, one of the key points is to efficiently exploiting SIMD vector units, another one is the memory access pattern. Thus, vectorization and optimization works have been conducted on a plasma turbulence application, namely Gysela. A set of different techniques have been used: loop splitting, inlining, grouping a set of LU solve operations, removing conditionals and some loop nests, auto-tuning of one computation kernel, changing a key numerical scheme – Lagrange interpolation instead of cubic splines. As a result, KNL execution times have been reduced by up to a factor 3 in some configurations. This effort has also permitted to gain a speedup of 2x on Broadwell architecture and 3x on Skylake. Nice scalability curves up to a few thousands cores have been obtained on a strong scaling experiment. Incremental work for vectorizing the Gysela code meant a large payoff without resorting to writing assembly code or using low-level intrinsics.

Keywords: Many-core, SIMD, KNL, vectorization, plasma physics

1 Introduction

The shrinking of computer components is still an ongoing trend and it is not yet limited by the laws of physics. Transistor size could be as low as 1 nm in 2033, as compared to 14 nm today[4]. Since 2004, another trend is a continuous increase of the amount of cores integrated on one chip. In the absence of a technological breakthrough, there are very few options available that can increase the performance of individual cores. One of these options turn out to be direct support for vector operations where a single instruction is applied to multiple data (SIMD). There are a range of alternatives for implementing vectorization, they vary in terms of complexity, flexibility, maintainability. In this paper, we will go through a set of techniques to automatically vectorize a large legacy code named Gysela. Shrinking trend is expected to continue with future many-core chips hosting larger core counts. Some problems are expected for the hardware: dealing with cache coherency at low cost, managing the contention on shared memory and network. We will not tackle these latter aspects here.

[4] Processors employing a 14 nm lithography process: Intel Skylake & Broadwell & KNL, AMD Ryzen.

The rest of this paper is organized as follows: reminder of Section 1 provides a description of some of the challenges offered by KNL hardware, our testbed, and the GYSELA application framework. Section 2 describes the initial status in terms of performance on one single node and three sets of improvements: some code transformations, algorithmic modifications and alternative numerical schemes are shown together with the impact on execution time. We show some strong scalings on multiple nodes in Section 3 on three clusters hosting different processors. Finally, we conclude in Section 4.

1.1 Many-core and KNL context

One option to improve single core performance is based on vector registers and SIMD instructions. SIMD operations exhibit parallelism proportional to the length of vector registers. Increasing vector length thus offers the opportunity to achieve speedups in codes through more SIMD parallelism. Some problems do however arise as the vector size increases. Branch mispredictions become very expensive and should be avoided. It becomes more difficult to regroup data between vector registers, to achieve efficient scatter/gather and masking operations, to deal with complex and/or irregular memory access patterns. This puts more pressure on compiler to select good optimization strategies. The observation of some rules of thumb in the code can however ease the compiler job. For example, in the innermost loops, one should ensure the number of iterations is larger than the vector length, restrict oneself to the set of available vector operations and rely on contiguous memory accesses. Given the current omnipresence of SIMD instructions, HPC codes must be optimized to exploit them. Current vector size in Intel KNL and Skylake is 512-bit, such kind of length is also expected soon in ARM processors (*c.f.* post-K computer, Japan). Larger register widths might be implemented, but one may wonder if this will really be profitable, except for very specific applications. Amdhal's law limits the benefit of SIMD as there always remains a fraction of the code that can not be vectorized. While longer vectors can improve performance they also have a cost. They complexify hardware design, require improvements in the compilers and lead to dependencies on the register width for the optimization process. In addition, many processors can not execute some SIMD instructions at their nominal frequency but a lower one, and a compromise has to be determined.

Actions have to be taken for the compiler to generate a proper executable with respect to vectorization. One can recall some of the classical techniques: loop fission, removal of conditionals and function calls from inner loops, algorithms transformation to signal the compiler the loops to vectorize, use of specific SIMD directives, introduction of new numerical schemes more suitable for vectorization, data alignment, removal of indirect memory accesses and promotion of unit-stride accesses, removal of loop-carried dependencies, strip mining using the vector width as blocking factor. Furthermore, using advanced profiling and performance analysis tools is helpful to get confidence in the quality of the vectorization. Large part of the optimizations targeting vectorization improve performance both on Intel KNL and on general-purpose multi-core architectures (as Haswell, Broadwell, Skylake).

Intel Knights Landing (KNL) is a standalone many-core processor. It has many features that one can exploit: a large number of threads, large vector units, multiple memory tiers, quite large memory bandwidth (MCDRAM). The chip provides up to 72 cores grouped in tiles, four threads per core, two levels of cache. MCDRAM is integrated on-package while DRAM is off-package and connected by six DDR4 channels. An on-die mesh interconnect keeps the full system coherent. Peak theoretical performance is announced at 6 Tflops of single precision and 3 Tflops of double

precision per KNL. It proposes several levels of NUMA exposure and variants for the cache coherence protocol that can be selected at boot time.

1.2 Testbed

During several months of 2017 we have had access to three partitions of the Marconi machine (Cineca's Tier-0 system in Bologna, Italy). These partitions hosted different processor architectures: Intel Broadwell, KNL and Skylake. We have been able to measure and compare performance there. The network Fabric is the same: Omnipath. Table 1 gives a brief summary of the hardware used.

Architecture (node)	Broadwell	KNL	Skylake
Nb cores	36=2x18	68	48=2x24
Vector register width	256-bits	512-bits	512-bits
Memory	128 GB	96 GB (DDR4) 16 GB (MCDRAM	192 GB
Frequency	2.3Ghz	1.4Ghz	2.1Ghz
AVX Frequency	2.0Ghz	1.2Ghz	1.8Ghz (AVX2) 1.4Ghz (AVX512)
FMA units	2	2	2
Peak TFlops/s (theory AVX freq.)	1.2	2.6	2.2
Memory Bandwidth GB/s	119	90 (DDR4) 490 (MCDRAM)	195
Power	2x145 W	230 W	2x150 W

Table 1: Characteristics of the nodes used for the benchmarks

1.3 Gysela setting

A key factor that determines the performance of magnetic plasma containment devices as potential fusion reactors is the transport of heat, particles, and momentum. To study turbulent transport and to model tokamak fusion plasmas, parallel codes have been designed over the years. Computational resources available nowadays have allowed the development of several petascale codes based on the well-established gyrokinetic framework. In this article, we focus on the GYSELA gyrokinetic code parallelized using a hybrid MPI/OpenMP paradigm[1,6,7,8]. The GYSELA code is a non-linear 5D global gyrokinetic full-f code which performs mainly flux-driven simulations of ion temperature gradient (ITG) driven turbulence. It solves the standard gyrokinetic equation for the full-f distribution function, *i.e.* no assumption on scale separation between equilibrium and perturbations is done. This 5D equation is self-consistently coupled to a 3D quasineutrality equation. The code also includes other features: ion-ion collisions, several kind of heat sources, kinetic electrons, multispecies capability.

Concerning the coordinate system, the three spatial dimensions are $\mathbf{x}_G = (r, \theta, \varphi)$ where r and θ are the polar coordinates in the poloidal cross-section of the torus, while φ refers to the toroidal angle. The velocity space has two dimensions: $v_\parallel$ being the velocity along the magnetic field lines and μ the magnetic moment corresponding to the action variable associated with the gyrophase. Let $z = (r, \theta, \varphi, v_\parallel, \mu)$ be a variable describing the 5D phase space. The time evolution of the ion guiding-center distribution function $\bar{f}(z)$ (main unknown) is governed by the gyrokinetic Boltzmann equation:

$$\partial_t \bar{f} + \frac{1}{B^*_{\parallel s}} \nabla_z \cdot \left(\frac{\mathrm{d}z}{\mathrm{d}t} B^*_{\parallel s} \bar{f} \right) = \mathcal{D}_r(\bar{f}) + \mathcal{K}_r(\bar{f}) + C(\bar{f}) + \mathcal{S}(\bar{f}) \tag{1}$$

where $\mathcal{D}_r$ and $\mathcal{K}_r$ are respectively a diffusion term and a Krook operator applied on a radial buffer region, C corresponds to a collision operator and $\mathcal{S}$ refers to source terms (see[6]). We solve this equation with a Strang splitting consisting in four 1D advections (along φ and $v_\parallel$ directions) and one 2D advection (r and θ directions are treated simultaneously), that are applied at each time step (see Algorithm 1).

The guiding-center motion described by the previous transport equation is coupled to a field solver (3D quasi neutral solver which is a Poisson-like solver, considering adiabatic response of electrons) that computes the electric potential $\phi(r, \theta, \varphi)$:

$$\frac{e}{T_e}(\phi - \langle\phi\rangle) = \frac{1}{n_0}\int J_0(\bar{f} - \bar{f}_{init})\,\mathrm{d}\mathbf{v} + \rho_i^2\nabla_\perp^2\frac{e\phi}{T_i} \tag{2}$$

Details about this last equation can be found in[5,6,8]. This Poisson-like equation gives the electric potential ϕ at each time step t depending on the main distribution function $\bar{f}$. One of the difficulties in the gyrokinetic approach is that the Boltzmann equation (1) deals with guiding-centers while the quasi-neutrality equation (2) acts on particles. The link between particles and guiding-centers is ensured via a gyroaverage operator. This gyroaverage operator will not be addressed in this paper but numerical details can be found in[3] and references herein. The derivatives of $J_0\,\phi$ along the toroidal direction are computed. Then, these quantities act as a feedback in the Boltzmann equation (1), they appear into the term $\frac{\mathrm{d}z}{\mathrm{d}t}B^*_{\parallel s}\bar{f}$ (see[6]). The Boltzmann solver represents the CPU critical part, *i.e.* usually more than 90% of computation time.

for time step $n \geq 0$ **do**
  Field solver, Derivative computation, Diagnostics
  Boltzmann solver:
    1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    Transpose of $\bar{f}$
    2D Advection in (r, θ) $(\forall(\mu, \varphi, v_\parallel) = [local], \forall(r, \theta) = [*])$
    Transpose of $\bar{f}$
    1D Advection in φ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    1D Advection in $v_\parallel$ $(\forall(\mu, r, \theta) = [local], \forall(\varphi, v_\parallel) = [*])$
    Sources, Krook, Diffusion operators

Algorithm 1: Sketch of the global GYSELA algorithm

The MPI domain decomposition is switched between advections as shown in Algorithm 1. The 4D distribution function (for a given μ value) is transposed just before and after the 2D advection along (r, θ). We use the following notation throughout the Algorithms of this paper: `local` indicates that in a given dimension each MPI process owns a parallel sub-domain, conversely * states that each MPI process possesses all points along a specified direction. Each processor exchange data to change its sub-domain from $(r = local, \theta = local, \varphi = *, v_\parallel = *, \mu = local)$ to a new sub-domain $(r = *, \theta = *, \varphi = local, v_\parallel = local, \mu = local)$. Even if it implies large communication volumes, this solution enables one to make use of standard 2D cubic spline over the whole domain $(r = *, \theta = *)$ and to remove a CFL-like condition[9]. Communications of the transpose step have good locality properties, the message exchanges are done inside a μ-communicator that groups together adjacent processes. During the 2013-2016 period, we obtained good strong scalings on different supercomputers with GYSELA, for example 60% efficiency at 65k cores on the curie machine[9] (Sandy-Bridge based), 87% on a BlueGene/Q machine at 32k cores[1]. However, vectorization and many-core issues have arisen since then, the following sections will present the adaptations required to accommodate the recent architectural changes.

2 Improving performance on a single node

2.1 Algorithms and performance status

The starting point of this paper is the state of the code before its optimization[9] for SMT (specifically Intel hyper-threading). These optimizations included enhancing the load balance of threads, removing some OpenMP and MPI synchronizations and increasing iteration loop count in many places to make the use of a large pool of threads efficient. We notably merged many adjacent OpenMP parallel regions into larger contiguous ones. Much idle time was removed this way. In addition, we were able to remove the assumption that the number of threads within one MPI process of GYSELA had to be a power of two. This was quite crucial to address a wide range of many-core systems.

2.2 Inlining, conditionals and loops

The first steps in adapting our code to KNL were to minimize the cost of function calls using inlining, to avoid or move away conditionals from the innermost loops and to reformulate some mathematical operations to ease the compiler work. Indeed some algorithms and/or programming styles can inhibit vectorization. Some of the requirements to help the vectorization process are: (i) the vectorized loop should be the innermost loop of a nest, (i) there should be no I/O nor function calls (apart from math functions) inside those loops, (iii) loop-carried and complex data dependencies should be avoided, (iv) the control flow should be uniform (exceptions exist but one should consider removing branches at first) and array notations should be promoted instead of pointers. A conditional branch is a control hazard, it introduces additional instructions, it can lead to pipeline stalls that can severely compromise the efficiency of the Vector Processing Unit (VPU).

Inlining Vectorization happens on the innermost loops that consist in simple enough code (typically a single block). It is therefore of the utmost importance to inline function calls in these loops and/or use specific pragmas to help the compiler auto-vectorization analysis (*e.g.* `!$omp declare simd` creates vectorized versions of a function that process vector arguments using SIMD). Practically, in GYSELA, we added inline functions (through the `!$dir force inline` directive and by moving the function declarations in header files) to help the auto-vectorization process.

Conditionals, loops We moved some conditionals lower in the call stack. Instead of having branch switches within the loops to be vectorized, we transformed the code in order to have them outside each innermost loop. In some routines of the code, there were several nested loops to provide a simple way to express a switch (see Fig. 1 lines 2-5). A set of more specialized routines have been introduced that avoids these nested loops while providing the same functionality. The switch between these specialized routines has been devolved to the caller.

Mathematical expressions and directives The SIMD instruction sets of processors tends to be less general than the scalar one. Specialized domain-specific operations are included, many operations are available only for some data types but not for others, and a high-level understanding of the computation is often required in order to take advantage of some of the features. In order to avoid going to assembly, the developer has to transform the code so that the auto-vectorization of the compiler achieve good optimizations. Some standard techniques we have used to transform our code include: 1) precompute and store some reciprocals (due to the large cost of divide on KNL this

is beneficial), 2) reformulate some mathematical expression and remove temporary scalar variables for simpler data dependencies analysis by the compiler 3) introduce small vectors as local variables (typically of the size of register width or a bit larger) together with strip-mining 4) add some explicit vectorization directives as `!$dir simd` or `!$dir ivdep`.

Sample code Fig. 1 exemplifies how the modifications described in this subsection have been applied. A conditional has been moved outside innermost loop and three loops were avoided thanks to the specialization of this code part (at many places, this code block is called enforcing a single iteration count for the `ir`, `itheta`, `ivpar` variables). A precomputation of one reciprocal is done and stored into the `product` variable. The main loop does not embed any temporary variables and the directive `!$dir ivdep` was added. These two facts help the auto-vectorization.

```
icount = 0
do ivpar = begin_dim4,end_dim4
 do iphi = begin_dim3,end_dim3
  do itheta = begin_dim2,end_dim2
   do ir = begin_dim1,end_dim1
    Bij = init_magnet%B_norm(ir,itheta)
    if (asktransp .and. transp_Bstar) then
     call precomp_transp_Bstar(ir,itheta,ivpar,Bs)
    else
     call precomp_Bstar(ir,itheta,ivpar,Bs)
    end if
    dPhidr_tmp = dPhidr_3D(ir,itheta,iphi)
    dPhidtheta_tmp = dPhidtheta_3D(ir,itheta,iphi)
    Br_tmp = init_magnet%Br(ir,itheta)
    Btheta_tmp = init_magnet%Btheta(ir,itheta)
    J_tmp = coord_sys%jacobian_space(ir,itheta)
    PoissBrack_Phi_phi = 1._F64/(J_tmp*Bij) * &
     (Br_tmp*dPhidtheta_tmp-Btheta_tmp*dPhidr_tmp)
    SvExB_gradphi(icount) = PoissBrack_Phi_phi/Bs
    icount                = icount+1
   end do
  enddo
 end do
end do
```

```
  ! Specialized version of the code given that ofen
  ! there is a single iteration in dimensions: 1,2,4
  ir     = begin_dim1
  itheta = begin_dim2
  ivpar  = begin_dim4

  ! conditional branch moved outside the main loop
  if (asktransp .and. transp_Bstar) then
    Bs = Bstar_PNrPNthNvpar(ir,itheta,ivpar)
  else
    Bs = Bstar_NrNthPNvpar(ir,itheta,ivpar)
  end if
  ! precomputation of a reciprocal to save compute time
  product = 1._F64/&
      (coord_sys%jacobian_space(ir,itheta)*&
      init_magnet%B_norm(ir,itheta)*Bs)
!DIR$ ivdep
  do iphi = begin_dim3, end_dim3
    SvExB_gradphi(iphi-begin_dim3) = &
        product * (init_magnet%Br(ir,itheta)*&
        dPhidtheta_3D(ir,itheta,iphi) - &
        init_magnet%Btheta(ir,itheta)*&
        dPhidr_3D(ir,itheta,iphi))
  end do
```

Fig. 1: Sketch of a code part of $E \times B$ compute (left-hand position), versus the new version of the same computations with inlining and branch removal (right-hand position)

Benchmark The best deployment that we have identified on KNL node is 4 MPI processes of 32 threads within a node of 68 cores (roughly two threads per core). The memory mode on KNL was set to cache and cluster mode to quadrant. On Marconi Broadwell and Skylake nodes, the hyper-threading support was unavailable, thus we imposed one thread per core and one process per processor (*i.e.* two processes per node). Table 2 summarizes the gains provided by techniques described in Sections 2.1 (load-balance enhancement) and 2.2 (improving vectorization and reducing low level overheads). The lines of the table focus on different operators of the application. Execution times are shown in seconds for a small test case. In percentage, the gain over the original version (prior to Section 2.1 modifications) is displayed. The improvements permit to decrease effectively execution times. The global execution times is reduced by -18% up to -41% depending on the machine. Even though we focused on improving the KNL performance, it turned out that all architectures took advantages of the changes.

2.3 Alternative interpolant and cache-friendly algorithm

Alternative interpolant High order methods require more floating point operations per degree of freedom than low order methods. One could expect high-order to slow down applications, but execution time is not directly proportional to computational cost. Increased operation efficiency

Step	Architecture Broadwell	KNL	Skylake
advec1D in $v_\parallel$	32.6 (-44%)	46.6 (-42%)	17.9 (-61%)
advec2D (r,θ)	28.3 (-31%)	34.7 (-18%)	16.0 (-46%)
transpose	31.2 (-25%)	13.0 (-51%)	15.6 (-53%)
heat source	9.7 (-13%)	17.3 (-23%)	6.1 (-25%)
diffusion in θ	10.4 (-13%)	10.7 (0%)	5.4 (-33%)
...			
Total	**196** (-22%)	**227** (-18%)	**120** (-41%)

Table 2: Breakdown of timing (in s) for a small GYSELA run. In parentheses, improvement brought by Sections 2.1 and 2.2 compared to the original version. Domain size is $256 \times 128 \times 64 \times 64 \times 1$.

Interpolant	Mem. load	store	Multiply	Add	Divide
1D spline	1	1	26	16	**1**
1D Lagrange 5^{th}	1	1	30	25	0
1D Lagrange 7^{th}	1	1	**48**	**37**	0
2D spline	1	1	60	40	**2**
2D Lagrange 5^{th}	1	1	**90**	**74**	0
2D Lagrange 7^{th}	1	1	**144**	**122**	0

Table 3: Estimates of the average costs associated to cubic spline versus Lagrange interpolants of degree 5^{th} and 7^{th} for a single interpolation. The context is a series of interpolations with good spatial and temporal localities in cache.

(*e.g* through good vectorization) can compensate the increase of computational cost. In addition, high-order schemes usually increase the achievable accuracy. Furthermore, the computation intensity for data in cache is large for high-order methods and this fits well with the idea that "FLOPS are almost free" in the Exascale landscape while costs associated to data accesses should increase. Finally, fewer degrees of freedom are required for a given accuracy level, which yields smaller stencil sizes and potentially less data movement (best suited for constraints foreseen for exascale).

In this context, we evaluated the benefits of 1D high-order Lagrange[2] interpolants instead of cubic splines[4,5]. Lagrange polynomials of degree 5 and 7 were selected. The degree 5 polynomials provide slightly lower accuracy than cubic splines within GYSELA runs, while the degree 7 polynomials are slightly more accurate than cubic splines. Tensor product was used in order to access 2D interpolations that are needed for 2D (r,θ) advections. Practically, splines require a set of coefficients that are computed prior to the interpolations. This step involves additional data moves and storage for the coefficients, but also extra operations (*i.e.* small LU systems to be solved) that the compiler have difficulties to vectorize. The Lagrange approach avoids these issues and the compiler is able to well vectorize the mathematical formulas. One remaining problem is the computational costs. They are given in Table 3 considering that one have a series of interpolations to perform (as in GYSELA context) and the cache is able (in average) to amortize the loads and stores down to one write, one read per interpolation. One has to mention, divide operation, which is located in the spline interpolation, behaves slowly compared to other basic math operations, especially on KNL. The number of multiplications and additions is larger for Lagrange than for splines, but as the vectorization is effective with Lagrange, the computational overhead is cleared as the performance results of Table 4 establish.

Cache-friendly one-strided advections Contiguous memory access patterns fit well with the SIMD approach. Many SIMD operations can reference aligned unit-stride vectors in-memory as part of the instruction, thus avoiding separate load/stores. In other words, contiguous accesses permit to save extra and possibly inefficient gather/scatter operations or strided load/store. To access memory efficiently, one has to favor inner loops with unit-stride, minimize indirect addressing, and align data to 64-byte boundaries on both KNL and Skylake. The compiler is able to perform specific vector optimizations if data is aligned to a certain byte boundary in memory. Some data layout transformations may help in that regard. Fig. 2 sketches a modified algorithm of the 1D $v_\parallel$ advection that diminishes long-strided accesses along the $v_\parallel$ dimension (`ivpar`). Instead of updating directly the main distribution function f over the last contiguous dimension (original

algorithm not shown), copies are performed in lines 6-9 and 26-30 to work on a temporary 2D tile. During the copy we mix the slowest varying index with the fastest varying one in order to benefit from fast reads from the main memory. Computations are then performed on the 2D tile, typically in L2 cache. Lines 13-22 copy data in ghost regions. This enables us to remove costly conditional branches related to boundary conditions along $v_\parallel$ in the main computations. This way, we have no conditional statements in the advection kernel (line 24), and we get better vectorization. All these modifications improve the quality of auto-vectorization and ensure cache-friendliness (use of L2 cache is improved and the TLB is less stressed by long-strided access). In addition, we added some SIMD directives to guide the compiler. Notice the 1D φ advection was modified in the same way.

Benchmark Table 4 exhibits timing obtained after the integration of the alternative Lagrange interpolant (5^{th} order) and the cache-friendly strategy plus the SIMD directives in 1D advections. In addition to the time values in Table 4, the reduction in percentage compared to those presented in the previous Table 2 is shown. Execution time of advections is greatly alleviated on all architectures.

```
1  !$OMP DO SCHEDULE(dynamic,1) collapse(2)
2   do iphi = 0, Nphi-1
3     do itheta = th_start, th_end
4       ! Copy from distrib function to tmp2d buffer
5       !   improve perf. because of contiguous access
6       do ivpar = 1, Nvpar-1
7         do ir = r_start, r_end
8           tmp2d(ivpar,ir) = f(ir,itheta,iphi,ivpar)
9         end do
10      end do
11      ! Boundary conditions with extra cells
12      !   avoid conditionals
13      do ivpar = -offset, 0
14        do ir = r_start, r_end
15          tmp2d(ivpar,ir) = f(ir,itheta,iphi,0)
16        end do
17      end do
18      do ivpar =  Nvpar, Nvpar+offset
19        do ir = r_start, r_end
20          tmp2d(ivpar,ir) = f(ir,itheta,iphi,Nvpar)
21        end do
22      end do
23      ! Perform advections in v//, update tmp2d(*,*)
24      ... Useful work here / vectorized ...
25      ! Copy back into distrib function
26      do ivpar = 0, Nvpar
27        do ir = r_start, r_end
28          f(ir,itheta,iphi,ivpar) = tmp2d(ivpar,ir)
29        end do
30      end do
31    end do
32  end do
```

Fig. 2: Sketch of the 1D advection along $v_\parallel$ with a copy that prevent long-strided accesses.

	Architecture		
Step	Broadwell	KNL	Skylake
advec1D in $v_\parallel$	13.0 (-60%)	12.8 (-73%)	6.6 (-63%)
advec2D (r,θ)	16.5 (-42%)	26.3 (-24%)	9.4 (-41%)
transpose	31.2	13.4	15.6
heat source	9.7	17.2	6.0
diffusion in θ	10.4	10.7	5.4
...			
Total	**146** (-26%)	**154** (-32%)	**90** (-25%)

Table 4: Breakdown of timing (in s) for a small run. In parentheses, improvement compared to the previous version (Table 2). Domain size $256 \times 128 \times 64 \times 64$.

	Architecture		
Step	Broadwell	KNL	Skylake
advec1D in $v_\parallel$	12.0 (-7%)	11.1 (-13%)	6.2 (-6%)
advec2D (r,θ)	7.2 (-46%)	6.9 (-74%)	4.1 (-56%)
transpose	30.9	13.1	15.5
heat source	4.3 (-56%)	3.6 (-79%)	2.3 (-62%)
diffusion in θ	4.1 (-60%)	3.3 (-69%)	2.6 (-52%)
...			
Total	**111** (-24%)	**89** (-42%)	**65** (-28%)

Table 5: Breakdown of timing (in s) for a small run. In parentheses, improvement compared to the previous version (Table 4). Domain size $256 \times 128 \times 64 \times 64$.

2.4 Additional vectorizing techniques

Vectorized LU solver Several routines of LAPACK can work with multiple right-hand-sides, and `dpttrs` in one of them. It solves a tridiagonal system $AX = B$ where X and B are general matrices and A is positive definite real symmetric. The computations performed by such routine is conceptually easy to transform into a set of SIMD instructions as the same steps are applied

at the same time to different right-hand-sides stored into small vectors. This can be achieved if caution is taken to well organize the storage of the right-hand-sides in memory. The developer has to carry out a data layout transform that may introduce a minor overhead, but it allows for a very efficient and vectorized implementation of the solve in the `dpttrs` routine. Fig. 3 displays the way we modified our code to benefit from a vectorized LU solver. Lines 6-8 precompute reciprocals in order to save many divide operations at line 18 within the solve. Lines 14-16 reorganize the storage layout such as the contiguous dimension correspond to vectorizing dimension. At line 18, we call an *ad hoc* version of `dpttrs` which is improved in order to take as input a vector of reciprocals (`Adiag_inv`) to dampen the cost of division on KNL. The vectorized LU solver is a algorithms that has been incorporated inside three operators: source, diffusion, derivation of the spline coefficients.

```
1  !*** Precomputation  ***
2  !*** A factorisation with LAPACK ***
3  Adiag(:)  = 4._F64
4  Aodiag(:) = 1._F64
5  call DPTTRF(n+1,Adiag,Aodiag,err)
6
7
8
9
10 !*** Many many times during execution
11 !*** Solving of x = (A^-1).rhs with LAPACK
12 do ...many times...
13   do i = 0, n
14     x(i) = rhs(...,i)
15   enddo
16
17
18   call DPTTRS(n+1,1,Adiag,Aodiag,x,n+1,err)
19 end do
```

```
1  !*** Precomputation  ***
2  !*** A factorisation with LAPACK ***
3  Adiag(:)  = 4._F64
4  Aodiag(:) = 1._F64
5  call DPTTRF(n+1,Adiag,Aodiag,err)
6  do i = 0, n
7    Adiag_inv(i) = 1.0/Adiag(i)
8  enddo
9
10 !*** Many many times during execution
11 !*** Solving of x = (A^-1).rhs
12 do ...many times...
13   do i = 0, n
14     do k = 0, nrhs-1
15       x(k,i) = rhs(...,k,i)
16     enddo
17   end do
18   call DPTTRS_modified(n+1,nrhs,Adiag_inv,Aodiag,x)
19 end do
```

Fig. 3: Sketch of a code part with tridiagonal solves with multiple right-hand sides (left-hand position), versus the new version vectorizing over the right-hand sides (right-hand position)

```
1  !$OMP DO SCHEDULE(static)
2      do ivpar = 0,Nvpar
3        do iphi = 0,Nphi-1
4          do itheta = th_start,th_jend
5            do ir = r_start, r_end
6              Sce_mu_tmp  = Sce_mu(ir,itheta,ivpar)
7              dSdx_mu_tmp = dSdx_mu(ir,itheta,ivpar)
8
9              dxdt = v_gradx(ir,itheta,iphi,ivpar)
10
11             deltaf = dt*Sce_mu_tmp - dt**2 * &
12                      (dxdt*dSdx_mu_tmp)
13             f(ir,itheta,iphi,ivpar) += deltaf
14           end do
15         end do
16       end do
17     end do
```

```
1  !$OMP DO SCHEDULE(dynamic, 1), collapse(2)
2      do ivpar = 0, Nvpar
3        do itheta = th_start,th_end
4  !DIR$ IVDEP
5          do ir = r_start, r_end
6            Sce_mu_vec(ir) = Sce_mu(ir,itheta,ivpar)
7            dSdr_mu_vec(ir)= Sdr_mu(ir,itheta,ivpar)
8          end do
9          do iphi = 0, Nphi-1
10 !DIR$ IVDEP
11           do ir = istart, iend
12             dxdt = v_gradx(ir,itheta,iphi,ivpar)
13             deltaf = dt*Sce_mu_vec(ir) - dt**2 * &
14                      (dxdt*dSdr_mu_vec(ir))
15             f(ir,itheta,iphi,ivpar) += deltaf
16           end do
17         end do
18       end do
```

Fig. 4: Sketch of a code part of source operator (left-hand position), versus the new version of the same computations with loop interchange and fission (right-hand position)

Loop Fission *Loop fission* (also named loop distribution) consists in splitting a single loop into more than one, generally to remove or simplify dependencies. It attempts to build simpler loop bodies (part of the original one) while keeping the same index range. One expects to ease the job of the compiler concerning dependency's analysis and isolate parts of the loop which inhibit vectorization (in addition one reduces the pressure on the vector registers).
In this manner, automatic vectorization is enhanced. This technique has been applied in several innermost loops of the code. It also has been combined with *loop interchange* in order to move

vector loops in the innermost region, and to move loop carried dependencies or conditional branches outermost. Fig. 4 depicts these loop transformations that have been carried out on the source operator. We also introduced intermediate aligned vectors in this algorithm (lines 7-8), it permitted us to reorganize data while transferring from the memory to the cache (peeling the memory accesses from the loop along φ direction) and to have them aligned.

Strip-mining, auto-tuning We introduced strip-mining technique within some GYSELA's operators. Also, we introduced small vectors declared as local variables. Their size were set accordingly with the strip-mining segment, it allows us to get a better SIMD-encoding from the compiler. For the specific case of 2D advection operator which represents a major cost, this size was auto-tuned. We will not give details here, but other parameters were also taken into account for this procedure: different compilers, languages (C/Fortran), several types of inlining, *etc*. It turned out, that once the compiler is chosen, the most crucial parameter is the size. Practically, we determined this size through a set of standalone tests using the BOAST framework[10] for each hardware we considered.

Contiguous In Fortran, one can use `contiguous` keyword (Fortran 2008) to tell the compiler that dummy arguments of a routine will always be contiguous in memory. Thus, the compiler is able to generate more efficient code.

Benchmark Timing measurements are shown in Table 5. The `contiguous` keyword helped to gain a few percents everywhere. The strip-mining were employed in 2D advection with success. Loop fission and LU vectorization helped to shorten the execution time of heat source and diffusion computations. Vectorization were widely improved, therefore the KNL execution times on one node became less than those of one Broadwell node, which is a good result. Also, these optimizations were valuable for all three processors we focused on. If one compares the final timers (Table 5) to the original timers (not displayed in this paper), the time reductions are -56% on Broadwell, -68% on KNL and -68% on Skylake. The process to vectorize this application code has resulted in major performance leap. Incremental work meant a large payoff for GYSELA without resorting to writing assembly code or using low-level intrinsics. Finally, we end up with a speedup of 7× on advection operators on all processors, and a speedup of $2\times$ to $3\times$ for the global execution times of the test cases we focused on.

3 Strong scaling on clusters

We ran a strong scaling test displayed in Figs. 5 and 6 using a domain size $512\times256\times128\times128\times16$, which is close to a production case used by physicists. The execution time is shown in Fig. 5 depending on the number of cores. One can see the decomposition in term of CPU time for the different components of the code. Even after decreasing the advection costs by a factor 7× (for the computational portion), these advections remain the biggest part in the breakdown of timings. Field solver, diagnostics and collisions take the largest fraction of the remaining elapsed time.

On Fig. 6, the diffusion and collisions parts scale almost perfectly because they are composed of computations only, also well balanced between MPI processes, without any communication. Other parts involve a mix of computations and communications. As the work is well balanced thanks to a domain decomposition that dispatches equally the computations, the overheads come mainly from communication costs.

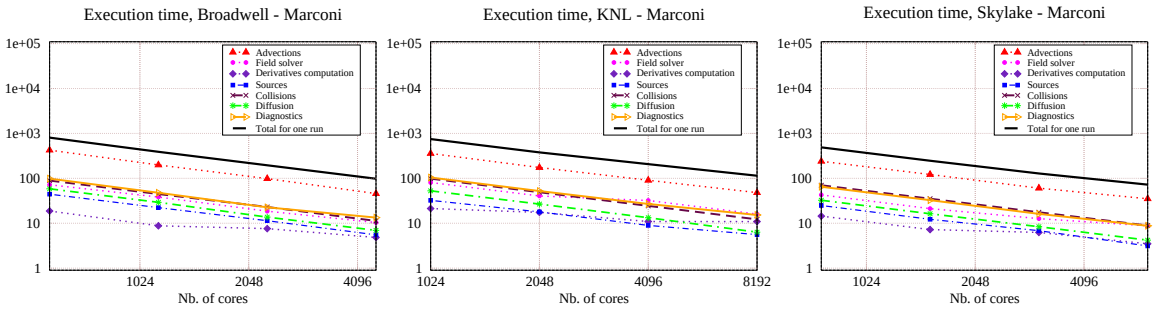

Fig. 5: Execution time from 16 to 128 nodes of a short Gysela run

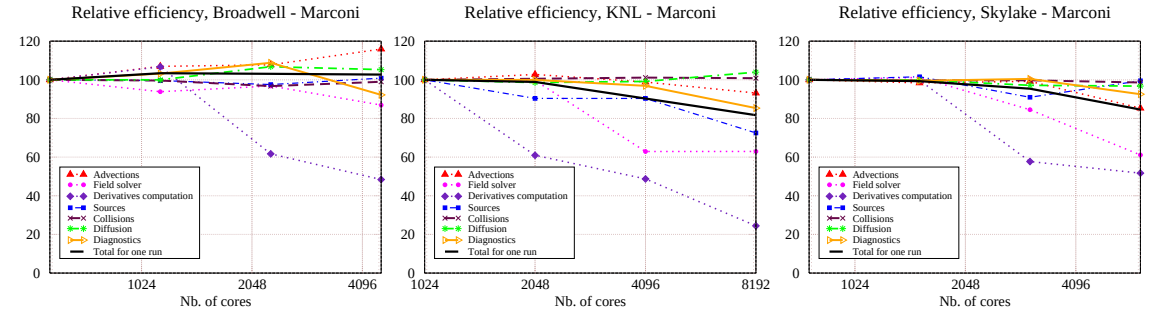

Fig. 6: Relative efficiency from 16 to 128 nodes of a short Gysela run

We tested a few experiments at larger scale on KNL partition (32k cores). We established that two components do not scale well: field solver and derivatives computation. They are characterized by many-to-many communication patterns and large data volumes exchanged[8,9]. In one test case, their cost was about 10 % to the total cost with 1k cores, but about 30% with 32k cores. Investigations are underway to find a remedy if possible. But, the continuously increasing gap between CPU speed and network bandwidth will make this task difficult.
Relative efficiency of the entire application considering 128 nodes reaches 103% on Broadwell (because of beneficial cache effects), 82% on KNL and 85% on Skylake. Execution time on 128 nodes are quite close for Broadwell and KNL, whereas Skylake performs much better.

4 Conclusion

Using both MPI and OpenMP parallelism was helpful to tune the GYSELA code for optimal performance on KNL and to fit the memory structure. We managed to have MPI processes that do not spread over more than one quadrant, which guaranteed uniform access to the memory and non-problematic cache and network behaviors. Good performance were achievable for the hybrid MPI+OpenMP code at medium scale (up to 128 nodes) without particular optimization of the MPI part. However, several OpenMP parts needed to be revised and prepared to welcome a large number of threads and hyper-threading (two threads per core was our best setting on KNL).

Most modern processors contain vector arithmetic units that benefit from the fine grain parallelism of vector operations. A large fraction of the peak performance originates from these vector

units. One can take advantage of these features through vectorizing compilers or by explicitly programming them with intrinsics (something we choose not to investigate for portability issues). In this paper, we have shown several manual transformations that can be applied to overcome compiler limitations and that allow for speedup through automatic vectorization. Namely, strip-mining, loop fission, inlining, transforming conditional branches and loops, SIMD directives are the techniques we employed to help the compiler to generate SIMD instructions. We also designed higher level approaches to reduce costs and shorten execution time. These include cache-friendly algorithms, high-order interpolants, transforming data layouts to use an efficient multiple right-hand side vectorized solver, and auto-tuning with the BOAST tool. Applying all these transformations, we achieved a speedup of 7× on the advection operators on all three architectures: KNL, Broadwell, Skylake. Furthermore, a speedup of $2\times$ to $3\times$ were observed on the global execution times of the test cases we focused on.

Strong scaling benchmarks show that performance behaves well up to a few thousands of cores. Relative efficiency stands in the range of 82% up to 103% on 128 nodes for the three hardware we considered. The field solver becomes a bottleneck at 32k cores on KNL partition and we are still investigating solutions to improve the involved communication schemes.

Acknowledgments *We benefited greatly from many fruitful discussions and advices from B. Pajot (Atos/Bull) and A. Farjallah (Intel). We would also like to stress that this work was supported by the Energy oriented Center of Excellence (EoCoE), grant agreement number 676629, funded within the EU's H2020 framework. We acknowledge GENCI for awarding us access to Frioul/CINES machine, especially G. Hautreux for support. Thank you to Eurofusion consortium and PRACE for using Marconi/CINECA partitions. The authors wish to commend C. Passeron for her constant help in the development of the Gysela code. This work has been carried out within the framework of the EUROfusion Consortium and has received funding from the Euratom research and training programme 2014-2018 under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.*

References

1. Bigot, J., Grandgirard, V., Latu, G., Passeron, C., Rozar, F., Thomine, O.: Scaling gysela code beyond 32K-cores on bluegene/Q. In: CEMRACS 2012. ESAIM: Proc., vol. 43, pp. 117–135. Luminy, France (2013)
2. Bouzat, N., Bressan, C., Grandgirard, V., Latu, G., Mehrenberger, M.: Targeting realistic geometry in Tokamak code Gysela. To be published - ESAIM: Proc. (2018), https://hal.archives-ouvertes.fr/hal-01653022
3. Bouzat, N., Rozar, F., Latu, G., Roman, J.: A new parallelization scheme for the hermite interpolation based gyroaverage operator. In: 16th Int. Symp. on Parallel and Distrib. Comp. (ISPDC). pp. 70–77 (2017)
4. Crouseilles, N., Latu, G., Sonnendrücker, E.: A parallel Vlasov solver based on local cubic spline interpolation on patches. Journal of Computational Physics 228, 1429–1446 (2009)
5. Grandgirard, V., et al.: A drift-kinetic Semi-Lagrangian 4D code for ion turbulence simulation. Journal of Computational Physics 217(2), 395 – 423 (2006)
6. Grandgirard, V., et al.: A 5d gyrokinetic full-f global semi-lagrangian code for flux-driven ion turbulence simulations. Computer Physics Communications 207, 35 – 68 (2016)
7. Latu, G., Crouseilles, N., Grandgirard, V., Sonnendrucker, E.: Gyrokinetic semi-Lagrangian parallel simulation using a hybrid OpenMP/MPI programming. In: Recent Advances in PVM and MPI, Lecture Notes in Computer Science, vol. 4757, pp. 356–364. Springer (2007)
8. Latu, G., Grandgirard, V., Crouseilles, N., Dif-Pradalier, G.: Scalable quasineutral solver for gyrokinetic simulation. In: PPAM (2). pp. 221–231. LNCS 7204, Springer (2011)
9. Latu, G., Bigot, J., Bouzat, N., Giménez, J., Grandgirard, V.: Benefits of SMT and of parallel transpose algorithm for the large-scale GYSELA application. In: PASC proc., Lausanne, June 8-10 (2016)
10. Videau, B., Pouget, K., Genovese, L., Deutsch, T., Komatitsch, D., Desprez, F., Méhaut, J.F.: BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. IJHPCA 32(1), 28–44 (2018)