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

Guillaume Latu, Yuuichi Asahi, Julien Bigot, Tamás Fehér, Virginie Grandgirard

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

Guillaume Latu
CEA, IRFM, FR-13108 St-Paul-lez-Durance

Yuuichi Asahi
QST Rokkasho Fusion Institute, Aomori, Japan

Julien Bigot
CEA, Maison de la Simulation, FR-91191 Gif-sur-Yvette

Tamas Feher
Max Planck Institute for Plasma Physics,
Garching, Germany

Virginie Grandgirard
CEA, IRFM,
FR-13108 St-Paul-lez-Durance

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. Many factors impact the performance achieved by applications on these devices: one of the key points is the efficient exploitation of SIMD vector units, and one another is the memory access pattern. Works have been conducted to adapt a plasma turbulence application, namely Gysela, for this architecture. A set of different techniques have been used: standard vectorization techniques, auto-tuning of one computation kernel, switching to high-order scheme. As a result, KNL execution times have been reduced by up to a factor 3. 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 meant a large payoff without resorting to using low-level intrinsics.**

Index Terms—**many-core, SIMD, vectorization**

I. 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[1]. 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 few options available that can increase the performance of individual cores. One of these options turns 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, which 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.

The rest of this paper is organized as follows: the remainder of Section I provides a description of some of the challenges offered by KNL hardware, our testbed, and the Gysela application. Section II describes the initial status in terms of performance on one single node and three sets of improvements: some code transformations, algorithmic & numerical scheme modifications are shown together with the impact on execution time. We show strong scalings in Section III on three clusters hosting different processors. Finally, we conclude in Section IV.

A. 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

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

the opportunity to achieve speedups in codes through more SIMD parallelism. Some problems do however arise as the vector size increases. Branch mispredictions become expensive. 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's 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. Current vector size in Intel KNL and Skylake is 512-bit. Amdahl'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. The complexity of hardware design requires improvements in the compilers and lead to dependencies on the register width for the optimization process. Also, many processors can not execute some SIMD instructions at their nominal frequency.

Actions have to be taken for the compiler to generate a proper executable with respect to vectorization. We will go through some of them in this paper. Furthermore, using advanced profiling and performance analysis tools is mandatory to get confidence in the quality of the vectorization. Most of the optimizations targeting vectorization improve performance both on KNL and on general-purpose multicore architectures.

Intel KNL is a standalone many-core processor. It has many features: a large number of threads, large vector units, multiple memory tiers, 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 interconnection keeps the full system coherent.

B. Testbed

We 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 I gives a brief summary of the hardware used. Intel processors adjust their frequency according to workload. Highly threaded, vectorized code may run in a lower frequency range.

Node architecture	Broadwell	KNL	Skylake
Nb cores	**36**=2x18	**68**	**48**=2x24
Vector reg. 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 BW GB/s	**119**	**90** (DDR4) **490** (MCDRAM)	**195**
Power	2x145 W	230 W	2x150 W

Table I: Architectures used for the benchmarks

C. 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. For this purpose, we need to study turbulent transport and to model tokamak fusion plasmas. In this article, we focus on an application that uses a hybrid MPI/OpenMP paradigm [1]. The Gysela code is a non-linear 5D global gyrokinetic full-f code which simulates turbulence driven by temperature gradient. 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. Let $\vec{z} = (r, \theta, \varphi, v_{\parallel}, \mu)$ be a variable describing the 5D phase space. The time evolution of the ion guiding-center distribution function $\bar{f}(\vec{z})$ (main unknown) is governed by the Boltzmann equation:

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

where $\mathcal{D}_r$ and $\mathcal{K}_r$ are respectively a diffusion term and a Krook operator applied on a radial buffer region, $\mathcal{C}$ corresponds to a collision operator and $\mathcal{S}$ refers to source terms (see [3] for more details). We solve this equation with a Strang splitting consisting in four 1D advections (along φ and $v_\|$ directions) and one 2D advection (r, θ directions are treated simultaneously), that are applied at each time step (see Algo. 1). The guiding-center motion described by the previous transport equation is coupled to a *field solver* (3D Poisson-like solver with adiabatic response of electrons) that computes the electric potential $\phi(r, \theta, \varphi)$:

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

Details about this last equation and the role of gyroaverage J_0 can be found in [3]. This Poisson-like equation gives the electric potential ϕ at each time step t depending on the main distribution function $\bar{f}$.

The MPI domain decomposition is switched between advections as shown in Algo. 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: `local` indicates that in a given dimension each MPI process owns a parallel subdomain, conversely `[*]` states that each MPI process possesses all points along a specified direction. During the 2013-2016 period, we obtained good strong scalings with this code, *e.g* 60% efficiency at 65k cores on a Sandy-bridge based machine, 87% on a BlueGene/Q machine at 32k cores. Details about the parallelization, reducing MPI costs, can be found in [1], [4]. However, vectorization and many-core issues have arisen since then. In the following we present adaptations to accommodate these recent architectural changes.

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

Algorithm 1: Sketch of the global Gysela algorithm

II. Improving performance on a single node

A. Load-balance and hyperthreading

In order to better use the multiple floating-point units available within the cores and to better support hyperthreading, we introduced a set of optimizations. These included enhancing the load balance of threads by means of more dynamic scheduling (see [4] for details). Then, we removed some OpenMP and MPI synchronizations. Also, we removed the assumption that the number of threads within one process had to be a power of two. This was crucial to address more systems.

B. Inlining, conditionals and loops

The first steps in adapting our code to KNL were to ease the compiler's work. Indeed some algorithms and/or programming styles inhibit vectorization. Some of the requirements to help the vectorization process are: (i) the vectorized loop should be the innermost loop of a nest, (ii) 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 compromise the efficiency of the Vector Processing Unit.

*a) **Inlining:*** Vectorization happens on the innermost loops that consist in simple enough code (typically a single block). It is then crucial to inline function calls in these loops (routines stored in different modules are not inlined by fortran compilers). We added inline functions (through the `!$dir force inline` directive and by moving the function declarations in header files). We also used specific pragmas to help the compiler auto-vectorization analysis (*e.g.* `!$omp declare simd` creates vectorized versions of a function that process vector arguments using SIMD).

b) ***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, first Algo., lines 2-5). A set of more specialized routines have been introduced that avoids these nested loops.The switch between these specialized routines has been devolved to the caller.

c) ***Expressions and directives:*** The SIMD instruction sets of processors tends to be less general than the scalar ones. Specialized domain-specific operations are included, many operations are available only for some data types, and a high-level understanding of the computation is often required in order to take advantage of them. 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 include: 1) precompute and store reciprocals (to avoid divisions), 2) reformulate some mathematical expressions and remove temporary variables for simpler data dependencies analysis, 3) introduce small vectors as local variables together with strip-mining 4) add explicit vectorization directives as `!$dir simd` (whenever auto-vectorization is insufficient).

d) ***Sample code:*** Fig. 1, second Algo., exemplifies some modifications described above. 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.

```
 1 icount = 0
 2 do ivpar = begin_dim4,end_dim4
 3  do iphi = begin_dim3,end_dim3
 4   do itheta = begin_dim2,end_dim2
 5    do ir = begin_dim1,end_dim1
 6     Bij = init_magnet%B_norm(ir,itheta)
 7     if (asktransp .and. transp_Bstar) then
 8      call precomp_transp_Bstar(ir,itheta,ivpar,Bs)
 9     else
10      call precomp_Bstar(ir,itheta,ivpar,Bs)
11     end if
12     dPhidr_tmp = dPhidr_3D(ir,itheta,iphi)
13     dPhidtheta_tmp = dPhidtheta_3D(ir,itheta,iphi)
14     Br_tmp = init_magnet%Br(ir,itheta)
15     Btheta_tmp = init_magnet%Btheta(ir,itheta)
16     J_tmp = coord_sys%jacobian_space(ir,itheta)
17     PoissBrack_Phi_phi = 1._F64/(J_tmp*Bij) * &
18      (Br_tmp*dPhidtheta_tmp-Btheta_tmp*dPhidr_tmp)
19     SvExB_gradphi(icount) = PoissBrack_Phi_phi/Bs
20     icount                = icount+1
21    end do
22   enddo
23  end do
24 end do
```

```
 1   ! Specialized version of the code given that often
 2   ! there is a single iteration in dimensions: 1,2,4
 3   ir     = begin_dim1
 4   itheta = begin_dim2
 5   ivpar  = begin_dim4
 6
 7   ! conditional branch moved outside the main loop
 8   if (asktransp .and. transp_Bstar) then
 9     Bs = Bstar_PNrPNthNvpar(ir,itheta,ivpar)
10   else
11     Bs = Bstar_NrNthPNvpar(ir,itheta,ivpar)
12   end if
13   ! precomputation of a reciprocal to save compute
       time
14   product = 1._F64/&
15       (coord_sys%jacobian_space(ir,itheta)*&
16       init_magnet%B_norm(ir,itheta)*Bs)
17 !DIR$ ivdep
18   do iphi = begin_dim3, end_dim3
19     SvExB_gradphi(iphi-begin_dim3) = &
20         product * (init_magnet%Br(ir,itheta)*&
21         dPhidtheta_3D(ir,itheta,iphi) - &
22         init_magnet%Btheta(ir,itheta)*&
23         dPhidr_3D(ir,itheta,iphi))
24   end do
```

Figure 1: Sketch of a code part of $E \times B$ compute (first algo.), versus the new version with inlining and branch removal (second algo.)

e) ***Benchmark:*** The best configuration that we have identified on a KNL node is 4 MPI processes of 32 threads within a node of 68 cores (roughly two threads per core with the hyperthreading). 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). Within a run, there are 8 iterations which means measures include an average over 8 samples. Table II summarizes the gains provided by techniques described above on one node (all cores used). The lines of the table focus on different operators of Gysela. Execution times are shown in seconds for a small case. In percentage, the gain over the original version is displayed. The global execution times is reduced by -18% up to -41% depending on the machine. It turned out that all architectures took advantages of the changes.

C. *High-order & cache-friendly algorithm*

*a) **Alternative interpolation**:* 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 (*e.g* through good vectorization) can compensate the increase of computational cost. 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.

In this context, we evaluated the benefits of 1D high-order Lagrange [2] interpolations instead of cubic splines [3]. The Lagrange polynomials of degree 5 provides close accuracy compared to cubic splines within Gysela runs. Tensor product was used in order to access 2D interpolations. Practically, splines require a set of coefficients that are computed prior to the interpolations. This step involves additional data moves, storage for the coefficients, but also extra operations (*i.e.* small LU systems to be solved) that the compiler have difficulties to vectorize. With the Lagrange approach, the compiler is able to well vectorize the simple mathematical formulas. One should also have a look to the computational costs. They are given in Table III considering that one have a series of interpolations to perform (Gysela context) and the cache is able (in average) to amortize the loads and stores down to one load, one store per interpolation. One has to mention, the divide operation, which is located in the spline interpolation, behaves slowly compared to other basic math operations 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 timings of Table IV establish.

	Architecture					
Step	Broadwell		KNL		Skylake	
advec1D in $v_{\|\|}$	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 II: Breakdown of timing (in s) for a small Gysela run. In parentheses, improvement brought by Sections II-A and II-B compared to the original version. Domain size is $256 \times 128 \times 64 \times 64 \times 1$.

	Mem.				
Interpolation \ Operation	load/store		×	+	/
1D spline	1	1	26	16	1
1D Lagrange 5th	1	1	30	25	0
2D spline	1	1	60	40	2
2D Lagrange 5th	1	1	90	74	0

Table III: Estimates of the average number of operations associated to cubic spline versus Lagrange interpolations for a single interpolation. Good spatial/temporal cache localities are assumed.

*b) **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 also to minimize indirect addressing, and to align data to 64-byte boundaries on both KNL and Skylake. Some data layout transformations may help in that regard.

Fig. 2 sketches a modified algorithm of the 1D $v_{\|}$ advection that diminishes long-strided accesses along the $v_{\|}$ 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 $v_{\|}$ in the main computations. This way, we have no conditional statements in the advection kernel (line 24). 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). The φ advection was modified in the same way.

*c) **Benchmark**:* Table IV exhibits timing obtained after improvements brought by Section II-C. In addition to these timings, the reduction in percentage compared to those presented in the previous Table II is shown. Execution time of advections is greatly alleviated on all architectures.

D. *Additional vectorizing techniques*

*a) **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 organizing 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. We modified our code to benefit from a vectorized LU solver (used into the heat source and diffusion operators).

*b) **Loop Fission**:* *Loop fission* (also known as loop distribution) consists in splitting a single loop into more than one, generally to

```
 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
```

Figure 2: Sketch of the 1D advection along $v_{\|}$ with a copy that prevent long-strided accesses.

	Architecture					
Step	Broadwell		KNL		Skylake	
advec1D in $v_{\|}$	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 IV: Breakdown of timing (in s) for a run. In parentheses, improvement compared to the previous version (Table II). Domain size $256 \times 128 \times 64 \times 64$.

	Architecture					
Step	Broadwell		KNL		Skylake	
advec1D in $v_{\|}$	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 V: Breakdown of timing (in s) for a small run. In parentheses, improvement compared to the previous version (Table IV). Domain size $256 \times 128 \times 64 \times 64$.

remove or simplify dependencies. It attempts to build simpler loop bodies (part of the original one) while keeping the same index range. This simplifies dependency analysis for the compiler and isolates the parts of the loop that inhibit vectorization (in addition this reduces the pressure on the vector registers). This technique has been applied in several innermost loops of the code. It has also 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. We also introduced intermediate aligned vectors, it permitted us to reorganize data while transferring from the memory to the cache and to have them aligned.

c) ***Strip-mining, auto-tuning:*** We introduced strip-mining technique within some Gysela's operators. We also introduced small vectors declared as local variables. Their *size* were set accordingly with the strip-mining segment, it enables 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 the most crucial parameter is the vector *size*. Practically, we determined this size through a set of tests using the BOAST framework [6] for each hardware.

d) ***Contiguous:*** In Fortran 2008, the `contiguous` keyword informs the compiler that dummy arguments of a routine will always be contiguous in memory thus enabling it to generate more efficient code.

e) ***Benchmark:*** Timing measurements are shown in Table V. The `contiguous` keyword helped gain a few percents everywhere. Strip-mining has been employed in the 2D advection with success. Loop fission and LU vectorization helped shorten the execution time of the heat source and diffusion computations. Vectorization has been widely improved, therefore the KNL execution time on one node is now less than on one Broadwell node, which is a good result (Table V). If one compares the final timers to the original ones, the time reductions reach -56% on Broadwell, -68% on KNL and -68% on Skylake. Improvements have resulted in a 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× to 3× for the global execution times of the considered test cases.

III. Strong scaling on clusters

We ran a strong scaling test displayed in Fig. 3 using a domain size 512×256×128×128×16, which is close to a production case used by physicists (other strong scalings in [1], [4]). From these efficiencies, one can see the diffusion and collisions parts scale almost perfectly. 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.

We tested a few experiments at larger scale on KNL partition (32k cores, not shown here). 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 [5]. 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. But, the continuously increasing gap between CPU speed and network bandwidth (a current trend in modern supercomputers) will make this task difficult.

Relative efficiency of the entire application considering 128 nodes reaches 103% on Broadwell, 82% on KNL and 85% on Skylake. The superlinear speedup is due to beneficial cache effects that largely compensate parallel overheads (the

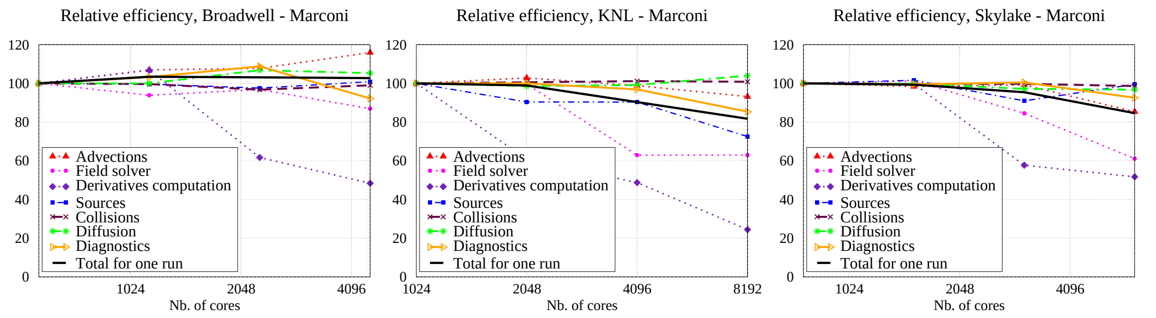

Figure 3: Efficiency up to 128 nodes of a short run, considered architectures : INTEL Broadwell, KNL and Skylake

amount of L3 cache is enlarged). Execution time on 128 nodes are quite close for Broadwell and KNL, whereas Skylake performs much better (in the range of [-40%:-25%]). We recorded close communication timings across the three architectures, though.

IV. Conclusion

A hybrid MPI/OpenMP approach was used for the Gysela code to get enhanced performance on many-core architectures. We managed to have MPI processes that do not spread over more than one quadrant of a KNL node, which guaranteed uniform access to the memory and non-problematic cache and network behaviors. Several OpenMP parts needed to be revised to welcome a large number of threads and hyper-threading.

In many modern processors, a large fraction of the peak performance originates from vector arithmetic 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. We have shown 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. We also designed higher level approaches to reduce costs and shorten execution time. These include cache-friendly algorithms, high-order interpolations, transforming data layouts to use an efficient multiple right-hand side vectorized solver, and auto-tuning. 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× to 3× were observed on the global execution times.

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 testbeds we considered.

Acknowledgments We benefited greatly from many fruitful discussions from B. Pajot (Atos/Bull) and A. Farjallah (Intel). This work was supported by the EoCoE project, grant agreement number 676629, funded within the EU's H2020. We acknowledge GENCI, PRACE for their machines. We received funding from the Euratom research and training programme under grant agreement No 633053. The views and opinions expressed herein do not necessarily reflect those of the European Commission.

References

[1] J. Bigot et al. Scaling gysela code beyond 32K-cores on bluegene/Q. In CEMRACS 2012, volume 43 of ESAIM: Proc., pages 117–135, Luminy, France, 2013.

[2] N. Bouzat, C. Bressan, V. Grandgirard, G. Latu, and M. Mehrenberger. Targeting realistic geometry in Tokamak code Gysela. To be published - ESAIM: Proc., 2018. https://hal.archives-ouvertes.fr/hal-01653022.

[3] V. Grandgirard et al. A 5D gyrokinetic full-f global semi-Lagrangian code for flux-driven ion turbulence simulations. Comp. Physics Comm., 207:35 – 68, 2016.

[4] G. Latu, J. Bigot, N. Bouzat, J. Giménez, and V. Grandgirard. Benefits of SMT and of parallel transpose algorithm for the large-scale GYSELA application. In PASC proc., Lausanne, June 8-10, 2016.

[5] G. Latu et al. Scalable quasineutral solver for gyrokinetic simulation. In PPAM (2), LNCS 7204, pages 221–231. Springer, 2011.

[6] B. Videau et al. BOAST: A metaprogramming framework to produce portable and efficient computing kernels for HPC applications. IJHPCA, 32(1):28–44, 2018.