Available on-line at www.prace-ri.eu

Partnership for Advanced Computing in Europe

XeonPhi Meets Astrophysical Fluid Dynamics

Evghenii Gaburov[a], Yuri Cavecchi[b]

[a]SURFsara, Science Park 140, 1098XG Amsterdam, the Netherlands
[b]Anton Pannekoek Institute, University of Amsterdam, the Netherlands

Abstract

This white paper reports on ours efforts to optimize a 2D/3D astrophysical (magento-)hydrodynamics Fortran code for XeonPhi. The code is parallelized with OpenMP and is suitable for execution on a shared memory system. Due to complexity of the code combined with immaturity of compiler we were unable to stay within the boundaries of Intel Compiler Suite. To deliver performance we took two different approaches. First, we optimized and partially rewrote most of the bandwidth-bound Fortran code to recover scalability on XeonPhi. Next, we ported several critical compute-bound hotspots to Intel SPMD Program Compiler (ISPC), which offers performance portability of a single source code across various architectures, such as Xeon, XeonPhi and possibly even GPU. This approach allowed us to achieve over 4x speed-up of the original code on dual-socket IvyBridge EP, and over 50x speed-up on the XeonPhi coprocessor. While the resulting optimized code can already be used in production to solve specific problems, we consider this project to be a proof-of-concept case reflecting the difficulty of achieving acceptable performance from XeonPhi on a "home-brewed" application.

Project ID: `2010PA1744/MagHydroBurn`

1. Introduction

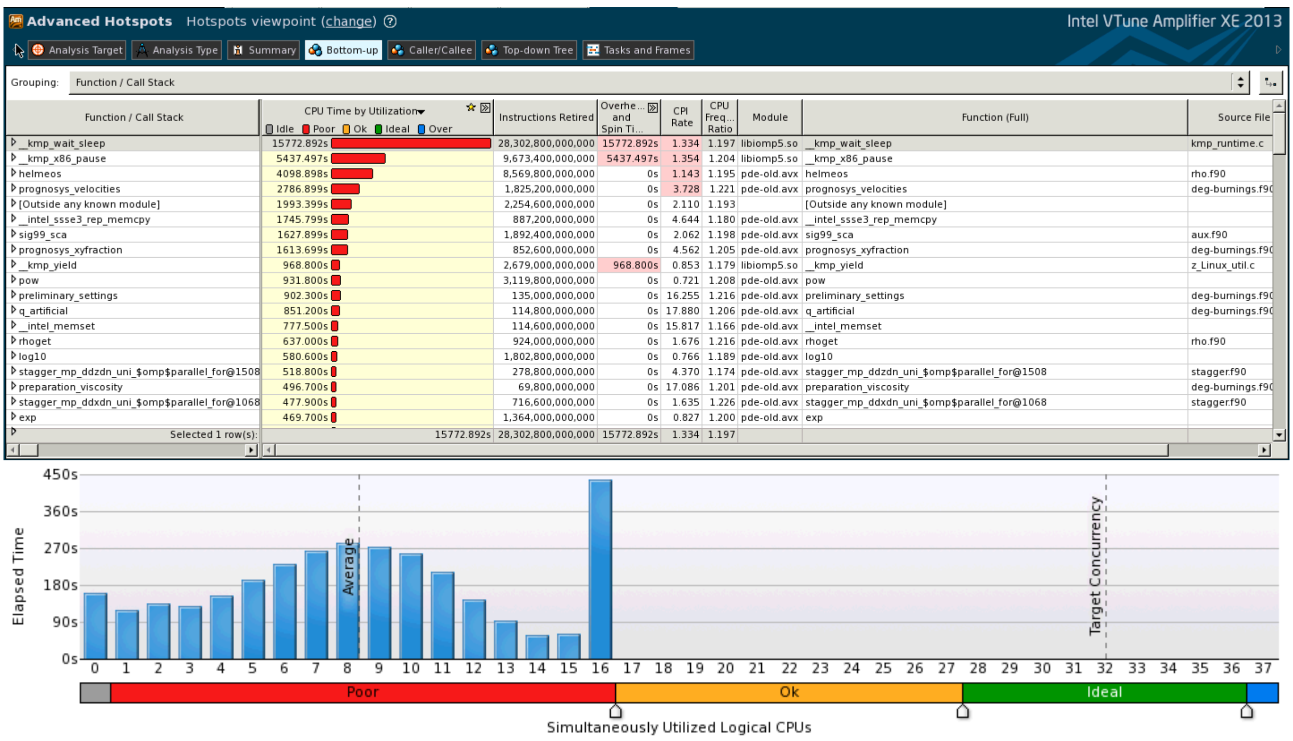
Accreting neutron stars have a thin layer of matter coming from a companion donor star. This layer, under the proper conditions, ignites and the resulting flame burns unstably propagating sideways via deflagration. This is a challenging astrophysical fluid dynamics problem due to vast range of length-scales that have to be resolved: from a few meters in the vertical direction to ∼10 kilometers in the horizontal direction. Furthermore, the problem also covers a wide range of time-scales: from ultra-fast nuclear burning reactions to sound-wave propagation. Additionally, the neutron star is rotating around its axis which requires inclusion of rotational effects, in particular the confining mechanism of Coriolis force is of crucial importance. We have already simulated, for the first time, deflagrating flames, while previous research was only focused on detonations, which is not what is believed to occur on the surface of neutron stars in nature.

Another important aim of this research is to understand the influence of the magnetic fields. For example, the magnetic field may confine the fluid by helping, or even substituting, Coriolis force, but it may also have negative effects on flame propagation. The hydrodynamic simulations, which initially were only carried out in 2D, must be extended to 3D in order to understand whether instabilities develop which can disrupt the flame due to the strong shear at its front. We also intend to simulate the propagation on the full geometry of the star, which may lead to more accurate interpretations and/or predictions of observations. Finally, the code is developed for any astrophysical configuration in which the fluid under analysis has an extreme aspect ratio, such as planet atmospheres or shells in stars.

In order to proceed with the aforementioned research plans, we need to make sure the application runs efficiently on the state-of-the art processors, such as Xeon, XeonPhi and GPU. This PRACE preparatory project was designed to have a closer look at the application and see what can be done to make the application run efficiently on XeonPhi. The resulting code optimizations will also be beneficial on Xeon, as well as the knowledge and experience, while not the source code, will be transferable to GPU should we wish to have a GPU port. The paper is structured as follows: In Section 2., we report on our analysis of the code and the resulting optimization strategies. The results and conclusions are presented in Sections 3. and 4. respectively.

2. Code description

The original programme is written in Fortran 90 and is parallelized with OpenMP for execution on shared memory systems. A quick runtime analysis on Xeon SandyBridge (SNB) and XeonPhi (KNC) processors


Advanced Hotspots Hotspots viewpoint (change)

Intel VTune Amplifier XE 2013

Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Instructions Retired	Overhead and Spin Time	CPI Rate	CPU Frequency Ratio	Module	Function (Full)	Source File
__kmp_wait_sleep	15772.892s	28,302,800,000,000	15772.892s	1.334	1.197	libiomp5.so	__kmp_wait_sleep	kmp_runtime.c
__kmp_x86_pause	5437.497s	9,673,400,000,000	5437.497s	1.354	1.204	libiomp5.so	__kmp_x86_pause	
helmeos	4098.898s	8,569,800,000,000	0s	1.143	1.195	pde-old.avx	helmeos	rho.f90
prognosys_velocities	2786.899s	1,825,200,000,000	0s	3.728	1.221	pde-old.avx	prognosys_velocities	deg-burnings.f90
[Outside any known module]	1993.399s	2,254,600,000,000	0s	2.110	1.193		[Outside any known module]	
__intel_ssse3_rep_memcpy	1745.799s	887,200,000,000	0s	4.644	1.180	pde-old.avx	__intel_ssse3_rep_memcpy	
sig99_sca	1627.899s	1,892,400,000,000	0s	2.062	1.198	pde-old.avx	sig99_sca	aux.f90
prognosys_xyfraction	1613.699s	852,600,000,000	0s	4.562	1.205	pde-old.avx	prognosys_xyfraction	deg-burnings.f90
__kmp_yield	968.800s	2,679,000,000,000	968.800s	0.853	1.179	libiomp5.so	__kmp_yield	z_Linux_util.c
pow	931.800s	3,119,800,000,000	0s	0.721	1.208	pde-old.avx	pow	
preliminary_settings	902.300s	135,000,000,000	0s	16.255	1.216	pde-old.avx	preliminary_settings	deg-burnings.f90
q_artificial	851.200s	114,800,000,000	0s	17.880	1.206	pde-old.avx	q_artificial	deg-burnings.f90
__intel_memset	777.500s	114,600,000,000	0s	15.817	1.166	pde-old.avx	__intel_memset	
rhoget	637.000s	924,000,000,000	0s	1.676	1.216	pde-old.avx	rhoget	rho.f90
log10	580.600s	1,802,800,000,000	0s	0.766	1.189	pde-old.avx	log10	
stagger_mp_ddzdn_uni_ompparallel_for@1508	518.800s	278,800,000,000	0s	4.370	1.174	pde-old.avx	stagger_mp_ddzdn_uni_ompparallel_for@1508	stagger.f90
preparation_viscosity	496.700s	69,800,000,000	0s	17.086	1.201	pde-old.avx	preparation_viscosity	deg-burnings.f90
stagger_mp_ddxdn_uni_ompparallel_for@1068	477.900s	716,600,000,000	0s	1.635	1.226	pde-old.avx	stagger_mp_ddxdn_uni_ompparallel_for@1068	stagger.f90
exp	469.700s	1,364,000,000,000	0s	0.827	1.200	pde-old.avx	exp	
Selected 1 row(s):	15772.892s	28,302,800,000,000	15772.892s	1.334	1.197			

Fig. 1. VTune screenshot displaying profiling (top) and concurrency (bottom) information for the original version of the code running on a single SNB socket (8 cores/16 threads). In the top panel, the first column shows function name, the second column the CPU time in seconds spent in it (approximately equal to runtime×#threads), the third column shows the number of instructions executed in the function with the clock-per-cycle (CPI) rate shown in column five. Finally, the fourth column shows overhead time which is the time spent in threading operations such as synchronization. The concurrency information in the bottom panel should be read as follows: the vertical axis is runtime in seconds, and bars show the wall-clock time the specific number threads (horizontal axis) were running simultaneously. Ideally, we would like to have the largest bar on the maximal number of threads available for the application; c.f. Fig. 3.

revealed a rather poor application performance[1]: the original application achieved $\lesssim 5\%$ and $\lesssim 0.5\%$ of the peak double precision performance on SNB and KNC respectively. A quick code analysis revealed that the application consist of hotspots that are somewhat evenly distributed between bandwidth- and compute-bound workload. To sum up, the initial analysis suggested that there is a potential for enormous speed up on KNC, and if achieved, the resulting optimizations will likely be of benefit for SNB as well.

2.1. Profiling with VTune

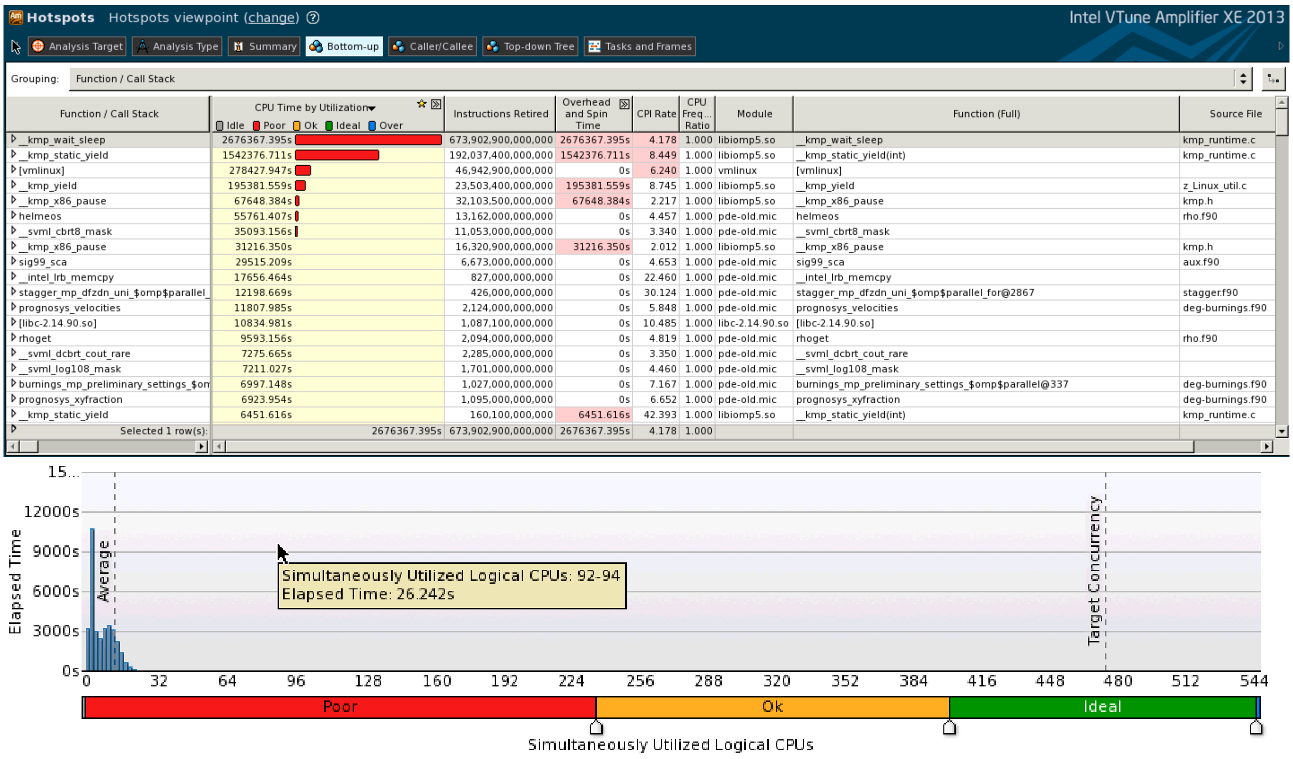
The first step was to profile the application, and for this task we used Intel VTune Amplifier (VTune). Fig. 1 and Fig. 2 display VTune screenshots from SNB and KNC runs respectively[2]. The application was run on a full chip in both cases, namely single socket SNB CPU, with 8 cores/16 threads (due to Intel HyperThreading Technology (HTT)), and all 60 cores/240 threads for KNC chip. We profile the code on a large problem size (1536x3840) that has sufficient amount of work for both XeonPhi and Xeon.

Few important observations can be made from the hotspot data: in both cases most of the runtime is spent in `__kmp_wait_sleep` and `__kmp_x86_pause` calls, which indicates severe load imbalance in application. This is also visually confirmed by the concurrency distribution shown below the hotspot information. In particular, on SNB about half of the time the number of active processing cores were less then half of the grand total available. With KNC, the concurrency distribution is fatal: $\lesssim$10% of processing cores are active. This is also confirmed by the KNC hotspot information, where $\gtrsim$80% of the runtime is spend in `__kmp_wait_sleep`. The data suggests that just by fixing the application concurrency we may be able to achieve up to ∼2x and ∼5x speed-up for SNB and KNC respectively.

In addition of identifying the application's load imbalance, the hotspot information revealed four hotspot functions that warrants closer inspection, we call these *the big four*: `helmeos`, `prognosys_velocities`, `sig99_sca` and `prognosys_xyfraction`; same list stands for KNC. We also noticed that substantial amount of time is spent in memory copies shown as (`__intel_ssse3_rep_memcpy` on SNB and `__intel_lrb_memcpy`, yet the application did not appear to have many explicit copies, thus warranting further investigation.

[1]We used `likwid-perctr` stethoscope mode from `https://code.google.com/p/likwid/` to measure application performance.

[2]Unless mentioned otherwise, we used Intel Compiler Suite 14.0.1 with "`-O3 -xavx -ipo -parallel -openmp`" and "`-O3 -mmic -ipo -parallel -openmp`" for SNB and KNC respectively.



Simultaneously Utilized Logical CPUs: 92-94
Elapsed Time: 26.242s

Function / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Instructions Retired	Overhead and Spin Time	CPI Rate	CPU Freq... Ratio	Module	Function (Full)	Source File
__kmp_wait_sleep	2676367.395s	673,902,900,000,000	2676367.395s	4.178	1.000	libiomp5.so	__kmp_wait_sleep	kmp_runtime.c
__kmp_static_yield	1542376.711s	192,037,400,000,000	1542376.711s	8.449	1.000	libiomp5.so	__kmp_static_yield(int)	kmp_runtime.c
[vmlinux]	278427.947s	46,942,900,000,000	0s	6.240	1.000	vmlinux	[vmlinux]	
__kmp_yield	195381.559s	23,503,400,000,000	195381.559s	8.745	1.000	libiomp5.so	__kmp_yield	z_Linux_util.c
__kmp_x86_pause	67648.384s	32,103,500,000,000	67648.384s	2.217	1.000	libiomp5.so	__kmp_x86_pause	kmp.h
helmeos	55761.407s	13,162,000,000,000	0s	4.457	1.000	pde-old.mic	helmeos	rho.f90
__svml_cbrt8_mask	35093.156s	11,053,000,000,000	0s	3.340	1.000	pde-old.mic	__svml_cbrt8_mask	
__kmp_x86_pause	31216.350s	16,320,900,000,000	31216.350s	2.012	1.000	libiomp5.so	__kmp_x86_pause	kmp.h
sig99_sca	29515.209s	6,673,000,000,000	0s	4.653	1.000	pde-old.mic	sig99_sca	aux.f90
__intel_lrb_memcpy	17656.464s	827,000,000,000	0s	22.460	1.000	pde-old.mic	__intel_lrb_memcpy	
stagger_mp_dfzdn_uni_ompparallel_	12198.669s	426,000,000,000	0s	30.124	1.000	pde-old.mic	stagger_mp_dfzdn_uni_ompparallel_for@2867	stagger.f90
prognosys_velocities	11807.985s	2,124,000,000,000	0s	5.848	1.000	pde-old.mic	prognosys_velocities	deg-burnings.f90
[libc-2.14.90.so]	10834.981s	1,087,100,000,000	0s	10.485	1.000	libc-2.14.90.so	[libc-2.14.90.so]	
rhoget	9593.156s	2,094,000,000,000	0s	4.819	1.000	pde-old.mic	rhoget	rho.f90
__svml_dcbrt_cout_rare	7275.665s	2,285,000,000,000	0s	3.350	1.000	pde-old.mic	__svml_dcbrt_cout_rare	
__svml_log108_mask	7211.027s	1,701,000,000,000	0s	4.460	1.000	pde-old.mic	__svml_log108_mask	
burnings_mp_preliminary_settings_$om	6997.148s	1,027,000,000,000	0s	7.167	1.000	pde-old.mic	burnings_mp_preliminary_settings_ompparallel@337	deg-burnings.f90
prognosys_xyfraction	6923.954s	1,095,000,000,000	0s	6.652	1.000	pde-old.mic	prognosys_xyfraction	deg-burnings.f90
__kmp_static_yield	6451.616s	160,100,000,000	6451.616s	42.393	1.000	libiomp5.so	__kmp_static_yield(int)	kmp_runtime.c
Selected 1 row(s):	2676367.395s	673,902,900,000,000	2676367.395s	4.178	1.000			

Fig. 2. VTune screenshot displaying profiling (top) and concurrency (bottom) information for the original version of the code running on a single KNC. For information about panels please refer to Fig. 1.

2.2. *Source code analysis and optimizations of bandwidth-bound workload*

After analysing *the big four* we concluded that two of them, `helmeos` and `sig99_sca`, are compute-bound, while the remaining two, as well as most of the code, are bandwidth-bound. While inspecting the source we also identified the cause of the excessive amount of memory copies. Specifically, the user application has dozens of calls to various functions that perform some category of stencil operations. While details of computations differ, all these functions bear an affinity to one another:

```
 1 ! function definition
 2 function foo(fin, ...)
 3     use params0     ! use a pre-defined parameter module
 4     implicit none
 5     real, dimension(mx,my,mz) :: foo
 6     real, dimension(mx,my,mz),intent(in) :: fin
 7     integer :: i,j,k
 8     ..
 9 !$omp parallel do private(i,j,k)
10     do k = 1,mz
11       do j = 1,my
12         do i = 1,mx
13           foo(i,j,k) = fin(i+1,j,k) + fin(i-1,j,k) + fin(i,j-1,k) + ...
14         end do
15       end do
16     end do
17 !$omp end parallel do
18 end function foo
19 ! function use
20 fres = foo(pos, ...);
```

List 1. Sample stencil function form in the original code

We would like to draw attention to the line 20 in List. 1. It is important to understand what the compiler does here in order to understand the source of the implicit copy operations. Every time `foo` is called, the compiler allocates a temporary 3D array on the stack frame (line 5), writes result into it (line 13), and then copies it to `fres` (line 20). This results in unnecessary read/write operation, just to copy results from temporal storage to `fres`. If data, as in the application at hand, doesn't fit the Last Level of cache (LL$), this can be heavily penalized. Indeed, consider the following simple change to the function:

```
1 !function defintion
2 subroutine foo(fin, ..., foo_out)
3     use params0     ! use a pre-defined parameter module
```

```
    implicit none
    real, dimension(mx,my,mz),intent(out) :: foo_out
    real, dimension(mx,my,mz),intent(in) :: fin
    integer :: i,j,k
    ..
!$omp parallel do private(i,j,k) schedule(dynamic)
    do k = 1,mz
      do j = 1,my
!dir$ ivdep
        do i = 1,mx
          foo_out(i,j,k) = fin(i+1,j,k) + fin(i-1,j,k) + fin(i,j-1,k) + ...
        end do
      end do
    end do
!$omp end parallel do
end subroutine foo
! function use
call (pos, ..., fres);
```
List 2. Sample stencil function form in optimized code

Here, we eliminate the unnecessary copy by passing **fres** directly to a subroutine. We also added dynamic scheduling to the OpenMP loop. This may have little effect on SNB, but it is of significant help on KNC due to larger core count sharing memory bus. This may easily lead to load imbalance among the threads, and the dynamic scheduling is (partially) able to rectify this source of imbalance.

In addition to these, we also replaced other unnecessary copies, such as

```
1   ...
2   call foo1(inp1, out1)
3   call foo2(inp2, out2)
4   scr1 = out1 + out2
5   call foo3(inp3, out3)
6   scr2 = scr1 * out3
7   res  = sqrt(scr2)
8   ..
```
List 3. Sample inefficient operations in the tuned code

with the following loops:

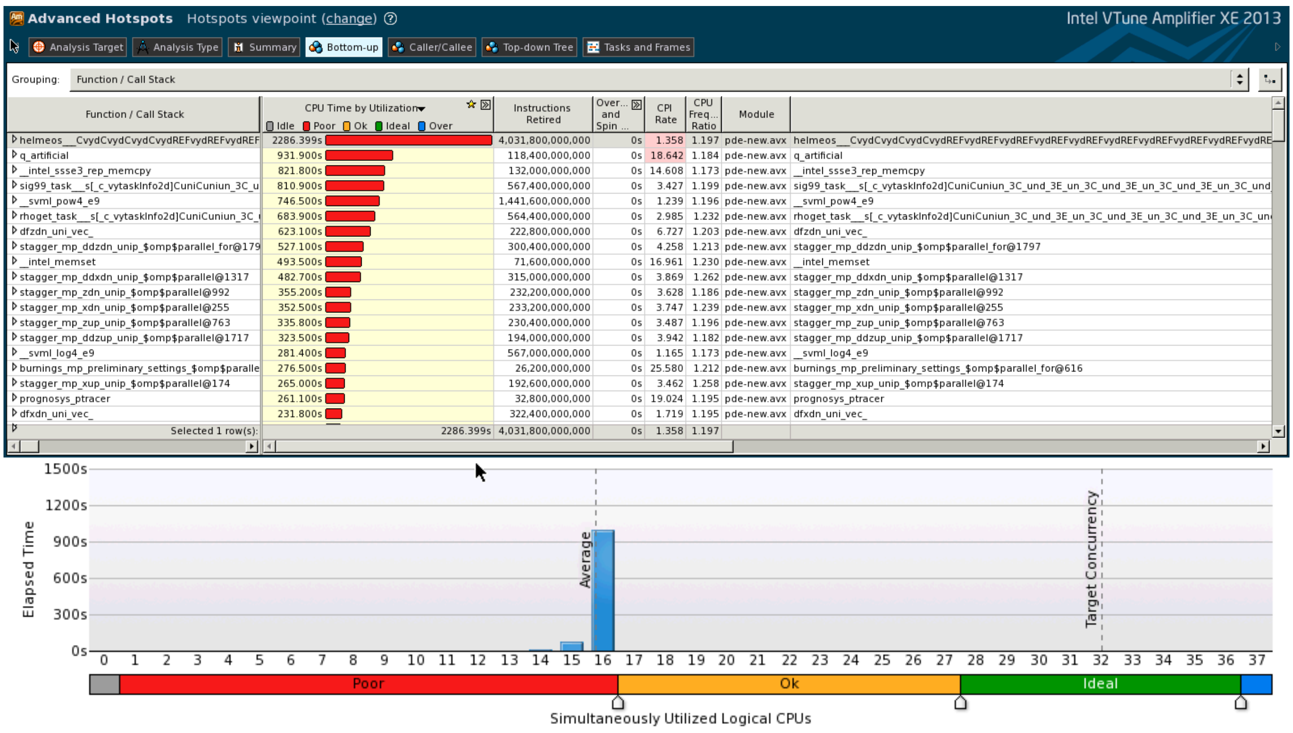
```
1    ...
2    call foo1(inp1, out1)
3    call foo2(inp2, out2)
4    call foo3(inp3, out3)
5  $!omp parallel do private(i,j,k,tmp1, tmp2) schedule(dynamic)
6    do k=1,mz
7      do j=1,m
8  !dir$ ivdep
9        do i=1,mx
10         tmp1 = out1(i,j,k) + out2(i,j,k)
11         tmp2 = tmp1 * out3(i,j,k)
12         res(i,j,k) = sqrt(tmp2)
13       end do
14     end do
15   end do
16 !$omp end parallell do
17   ...
```
List 4. Optimized operations in the tuned code

While this increases the number of code lines [3] , this optimization substantially decreases the number of implicit copy operations[4]. This decrease in the number of copies brings substantial performance benefits because the temporal results, which are originally stored in **scr1** and **scr2** arrays and are not eliminated by compiler (line 4 and line 6 in List. 3), are now explicitly stored either in registers or L1$ (line 10 and line 11 in List. 4). Furthermore, it appears to us that compiler "**-parallel**" flag does a poor job on KNC when it comes to copying large arrays, thus even a simple copy gained over 20% in efficiency from explicitly writing triple nested OpenMP loop with dynamic scheduling.

After we "patched" in this way nearly all parts of the code (including converting all relevant functions into subroutines as shown in List. 1 and 2) that we think mattered (circa 1k lines) we observed decrease in runtime by about 2x on SNB and substantial improvement in concurrency. While we don't have numbers at hand, we are confident that these optimizations are of even greater benefit on KNC. In particular, after this optimization step the wall-clock time spent in most of the bandwidth-bound workload paled in comparison to the two compute-bound functions: **helmeos** and **sig99_sca**, because the aforementioned optimizations have virtually zero influence on these two functions.

[3]This can be abstracted in C++, which will result in a compact code in comparison to Fortran or C equivalent.

[4]We would like to stress that the application was compiled with "**-parallel**" flag, which means all implicit copy operations *were* parallelized.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization (🟫 Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)	Instructions Retired	Over... and Spin ...	CPI Rate	CPU Freq... Ratio	Module	
helmeos__CvydCvydCvydCvydREFvydREFvydREF	2286.399s	4,031,800,000,000	0s	1.358	1.197	pde-new.avx	helmeos__CvydCvydCvydCvydREFvydREFvydREFvydREFvydREFvydREFvydREFvydREFvydRE
q_artificial	931.900s	118,400,000,000	0s	18.642	1.184	pde-new.avx	q_artificial
__intel_ssse3_rep_memcpy	821.800s	132,000,000,000	0s	14.608	1.173	pde-new.avx	__intel_ssse3_rep_memcpy
sig99_task__s[_c_vytaskInfo2d]CuniCuniun_3C_u	810.900s	567,400,000,000	0s	3.427	1.199	pde-new.avx	sig99_task__s[_c_vytaskInfo2d]CuniCuniun_3C_und_3E_un_3C_und_3E_un_3C_und_3E_un_3C_und
__svml_pow4_e9	746.500s	1,441,600,000,000	0s	1.239	1.196	pde-new.avx	__svml_pow4_e9
rhoget_task__s[_c_vytaskInfo2d]CuniCuniun_3C_	683.900s	564,400,000,000	0s	2.985	1.232	pde-new.avx	rhoget_task__s[_c_vytaskInfo2d]CuniCuniun_3C_und_3E_un_3C_und_3E_un_3C_und_3E_un_3C_un
dfzdn_uni_vec_	623.100s	222,800,000,000	0s	6.727	1.203	pde-new.avx	dfzdn_uni_vec_
stagger_mp_ddzdn_unip_ompparallel_for@179	527.100s	300,400,000,000	0s	4.258	1.213	pde-new.avx	stagger_mp_ddzdn_unip_ompparallel_for@1797
__intel_memset	493.500s	71,600,000,000	0s	16.961	1.230	pde-new.avx	__intel_memset
stagger_mp_ddxdn_unip_ompparallel@1317	482.700s	315,000,000,000	0s	3.869	1.262	pde-new.avx	stagger_mp_ddxdn_unip_ompparallel@1317
stagger_mp_zdn_unip_ompparallel@992	355.200s	232,200,000,000	0s	3.628	1.186	pde-new.avx	stagger_mp_zdn_unip_ompparallel@992
stagger_mp_xdn_unip_ompparallel@255	352.500s	233,200,000,000	0s	3.747	1.239	pde-new.avx	stagger_mp_xdn_unip_ompparallel@255
stagger_mp_zup_unip_ompparallel@763	335.800s	230,400,000,000	0s	3.487	1.196	pde-new.avx	stagger_mp_zup_unip_ompparallel@763
stagger_mp_ddzup_unip_ompparallel@1717	323.500s	194,000,000,000	0s	3.942	1.182	pde-new.avx	stagger_mp_ddzup_unip_ompparallel@1717
__svml_log4_e9	281.400s	567,000,000,000	0s	1.165	1.173	pde-new.avx	__svml_log4_e9
burnings_mp_preliminary_settings_ompparalle	276.500s	26,200,000,000	0s	25.580	1.212	pde-new.avx	burnings_mp_preliminary_settings_ompparallel_for@616
stagger_mp_xup_unip_ompparallel@174	265.000s	192,600,000,000	0s	3.462	1.258	pde-new.avx	stagger_mp_xup_unip_ompparallel@174
prognosys_ptracer	261.100s	32,800,000,000	0s	19.024	1.195	pde-new.avx	prognosys_ptracer
dfxdn_uni_vec_	231.800s	322,400,000,000	0s	1.719	1.195	pde-new.avx	dfxdn_uni_vec_
Selected 1 row(s):	2286.399s	4,031,800,000,000	0s	1.358	1.197		

Fig. 3. VTune screenshot displaying profiling (top) and concurrency (bottom) information for the optimized version of the code running on a single SNB socket. For information about panels please refer to Fig. 1.

2.3. *Tuning compute bound functions for XeonPhi*

The XeonPhi programming manual makes it very clear that in order to harvest the computing power of the coprocessor requires the user application to be (auto-)vectorizable: nearly 90% of peak double precision performance comes from wide-vector unit (up-to 8 double precision floating points per math operation). Thus our first task was to inspect what prevents Intel Fortran Compiler to auto-vectorize the two compute-bound functions: `helmeos` and `sig99_sca`. The `helmeos` function is called from `rhoget` function which is a custom implementation of Newton-Raphson iterative root-finding algorithm, in which `helmeos` is called from within branches of `if`-statements. Without going too much into details, it became very clear from inspection of the function sources and compiler reports that the code is too complex for Intel Fortran Compiler to auto-vectorize, and the only remaining route is manual vectorization[5].

The first attempt was made with the help of `!dir$ simd` but the code logic, which includes nested branching and calls to `helmeos` inside the branches, is again too complicated for the compiler to vectorize this even with `!dir$ simd`. However, as a proof-of-concept, we were able to rewrite this functions in AVX intrinsics which demonstrated $\gtrsim$2x speed-up for `helmeos` and `sig99_sca` respectively. This lead us to conclude that these two functions can be and *must* be manually vectorized for efficient execution on XeonPhi. While intrinsics is a fine approach when the source code will not require subsequent modification or is tailored for a particular hardware, the portability and expandability requirement did not permit us to ship intrinsics version of the code to the user; in other words, there would simply be no way for the user to easily modify the code without spending valuable time to learn intrinsics.

For this very reason we decided to use Intel SPMD Program Compiler, ISPC in short, in which to rewrite `rhoget`, `helmeos` and `sig99_sca`[6]. For performance comparison we timed the runtime of both intrinsics and ISPC versions of the functions[7]. This benchmarking showed that ISPC version was at most ∼10% slower than the intrinsics version–the price we are willing to pay in exchange for both portability and expandability.

The big anticipated bonus from this enterprise was that the same ISPC source code for ports of these three functions was also able to deliver speed-up of ∼3-4x on KNC. In the process of identifying the cause of less than expected speed-up, we stumbled upon a potentially serious performance bug in Intel C Compiler[8]. The bug has been reported to Intel, but as of version 14.0.1 it is not fixed; although we do expect measurable performance improvement once the bug is fixed, we did not investigate it close enough to estimate by how much performance will be improved.

[5]We refer the interested reader to inspect `sig99` function in `http://cococubed.asu.edu/code_pages/kap.shtml` from which `sig99_sca` is derived.

[6]See `http://ispc.github.com`. However, at the time of our porting activities, the KNC support in ISPC was incomplete for our purpose. This motivated us to fix this, and as of version 1.5.0 the support of KNC inside the ISPC is complete.

[7]We used `-xavx` and `--target=avx1-i64x4` compiler flags for intrinsics and ISPC functions.

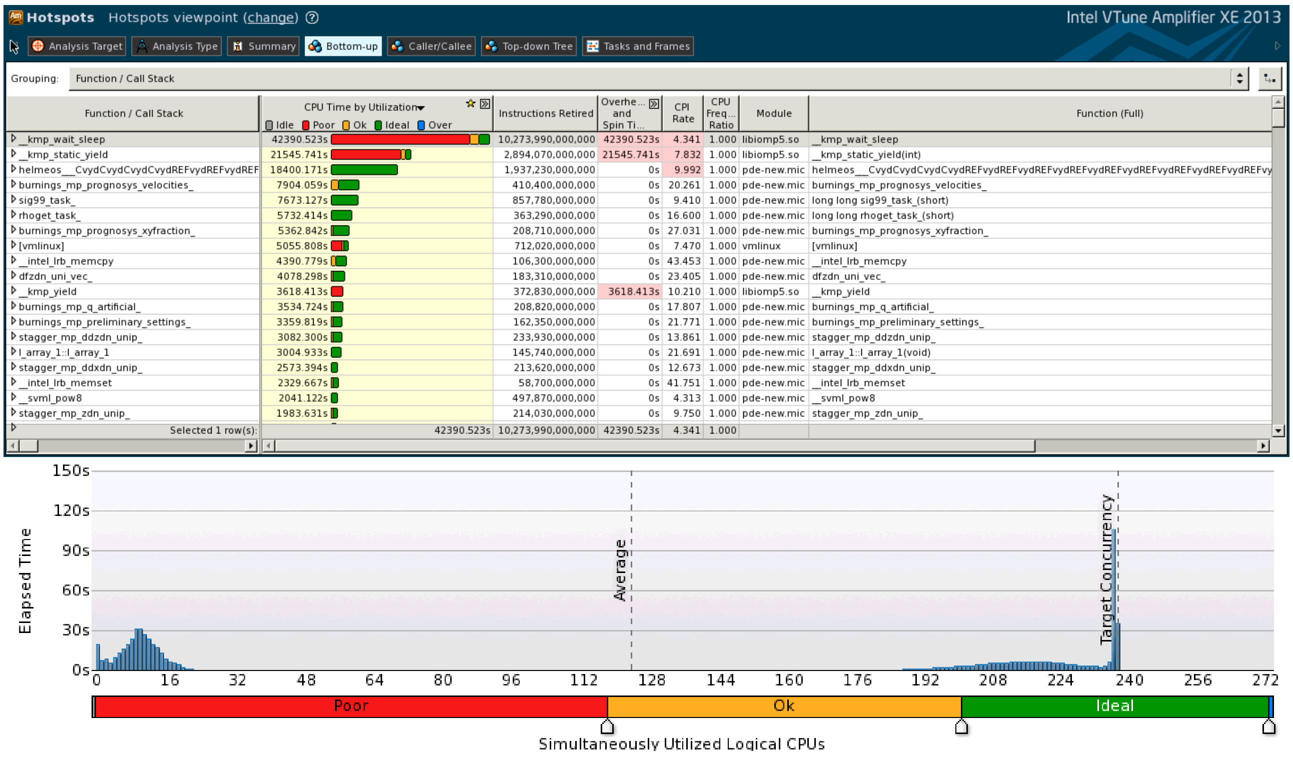
[8]The compilation process for KNC with ISPC is the following: source-to-source with ISPC (`.ispc`→`.cpp`) and source-to-object with ICC (`.cpp`→`.o`).

2.4. Profiling optimized code

We conclude this section by profiling the optimized code. In Fig. 3 and Fig. 4 we show hotspot lists and concurrency distributions for SNB and KNC respectively. From the bottom panel of Fig. 3 it becomes clear that the code has excellent concurrency on Xeon CPU, and from the top panel we can see that the load imbalance is nearly gone with about 20% of runtime being spent in `helmeos`. There are still memory copies present, but their source is known to us and we are convinced that further optimization will produce diminishing returns compared to increase in the code complexity.

We find it interesting to compare the number of instruction retired for compute-bound function. Comparison of top-panels in Fig. 1 and Fig. 3 shows that on SNB the optimized function executed 2.2 and 3.3 fewer instructions for `helmeos` and `sig99_sca` respectively. In case of KNC, however, the number of instructions is reduced by 6.8 and 7.8 for `helmeos` and `sig99_sca`. Given that both functions are dominated by double precision mathematical computations, and that KNC can execute 8 double-precision floating points per instruction, the near 8x reduction is in line with expectations. The speed-up, however, is more modest: 3.0x and 3.8x respectively. The CPI column shows that this is due to doubling amount of cycles spent per instruction on average. The reason for this is unknown to us at the moment, and must be investigated further.

Finally, we explore the concurrency of the code on XeonPhi from the bottom panel of the Fig. 4. When compared to the original concurrency distribution in Fig. 2, the improvement is phenomenal. Still, about 40% of runtime is spent in less than ∼20 threads (∼5 cores). It requires further investigation to understand what causes this. The solution to the remaining concurrency problem has a potential to deliver up to 1.6x speed-up, which certainly warrants the effort. Combined with the possibility to recover original CPI for `helmeos` and `sig99_sca`, we are looking at an additional ∼2x speed-up for the whole application on KNC.



Grouping: Function / Call Stack

Function / Call Stack	CPU Time by Utilization (Idle, Poor, Ok, Ideal, Over)	Instructions Retired	Overhe... and Spin Ti...	CPI Rate	CPU Freq... Ratio	Module	Function (Full)
__kmp_wait_sleep	42390.523s	10,273,990,000,000	42390.523s	4.341	1.000	libiomp5.so	__kmp_wait_sleep
__kmp_static_yield	21545.741s	2,894,070,000,000	21545.741s	7.832	1.000	libiomp5.so	__kmp_static_yield(int)
helmeos___CvydCvydCvydCvydREFvydREFvydREF	18400.171s	1,937,230,000,000	0s	9.992	1.000	pde-new.mic	helmeos___CvydCvydCvydCvydREFvydREFvydREFvydREFvydREFvydREFvydREFvydREFvy
burnings_mp_prognosys_velocities_	7904.059s	410,400,000,000	0s	20.261	1.000	pde-new.mic	burnings_mp_prognosys_velocities_
sig99_task_	7673.127s	857,780,000,000	0s	9.410	1.000	pde-new.mic	long long sig99_task_(short)
rhoget_task_	5732.414s	363,290,000,000	0s	16.600	1.000	pde-new.mic	long long rhoget_task_(short)
burnings_mp_prognosys_xyfraction_	5362.842s	208,710,000,000	0s	27.031	1.000	pde-new.mic	burnings_mp_prognosys_xyfraction_
[vmlinux]	5055.808s	712,020,000,000	0s	7.470	1.000	vmlinux	[vmlinux]
__intel_lrb_memcpy	4390.779s	106,300,000,000	0s	43.453	1.000	pde-new.mic	__intel_lrb_memcpy
dfzdn_uni_vec_	4078.298s	183,310,000,000	0s	23.405	1.000	pde-new.mic	dfzdn_uni_vec_
__kmp_yield	3618.413s	372,830,000,000	3618.413s	10.210	1.000	libiomp5.so	__kmp_yield
burnings_mp_q_artificial_	3534.724s	208,820,000,000	0s	17.807	1.000	pde-new.mic	burnings_mp_q_artificial_
burnings_mp_preliminary_settings_	3359.819s	162,350,000,000	0s	21.771	1.000	pde-new.mic	burnings_mp_preliminary_settings_
stagger_mp_ddzdn_unip_	3082.300s	233,930,000,000	0s	13.861	1.000	pde-new.mic	stagger_mp_ddzdn_unip_
l_array_1::l_array_1	3004.933s	145,740,000,000	0s	21.691	1.000	pde-new.mic	l_array_1::l_array_1(void)
stagger_mp_ddxdn_unip_	2573.394s	213,620,000,000	0s	12.673	1.000	pde-new.mic	stagger_mp_ddxdn_unip_
__intel_lrb_memset	2329.667s	58,700,000,000	0s	41.751	1.000	pde-new.mic	__intel_lrb_memset
__svml_pow8	2041.122s	497,870,000,000	0s	4.313	1.000	pde-new.mic	__svml_pow8
stagger_mp_zdn_unip_	1983.631s	214,030,000,000	0s	9.750	1.000	pde-new.mic	stagger_mp_zdn_unip_
Selected 1 row(s):	42390.523s	10,273,990,000,000	42390.523s	4.341	1.000		

Fig. 4. VTune screenshot displaying profiling (top) and concurrency (bottom) information for the optimized version of the code running on a single KNC. For information about panels please refer to Fig. 1.

3. Results

We present the final results on different processors and different problem sizes in Fig. 5. Here, the horizontal axis represents hardware on which the application was run, the vertical axis is the runtime in seconds, and we use color-coding for different problems size with the corresponding legend on the top left corner of the figure. For the reader's convenience we also print the run-time on each of the bars. In contrast to the previous sections, we collected the runtime information for this plot on compute node equipped with a dual-socket Ivy Bridge E5-2695 v2 with HTT enabled (IVB), which has a total of 24 cores/48 threads available for computation. This gives a fair comparison between different state-of-the-art processors from Intel.

Comparison of peak performances and peak bandwidth shows XeonPhi 5100 is about 2x faster than such a dual-socket IVB node. This number should anchor our expectations for the runtime numbers, namely we should not see more than 2x speed-up on XeonPhi. If this was the case, the only explanation would be that the Xeon code is not optimized, which is a highly unlikely proposition because most of optimizations for XeonPhi are

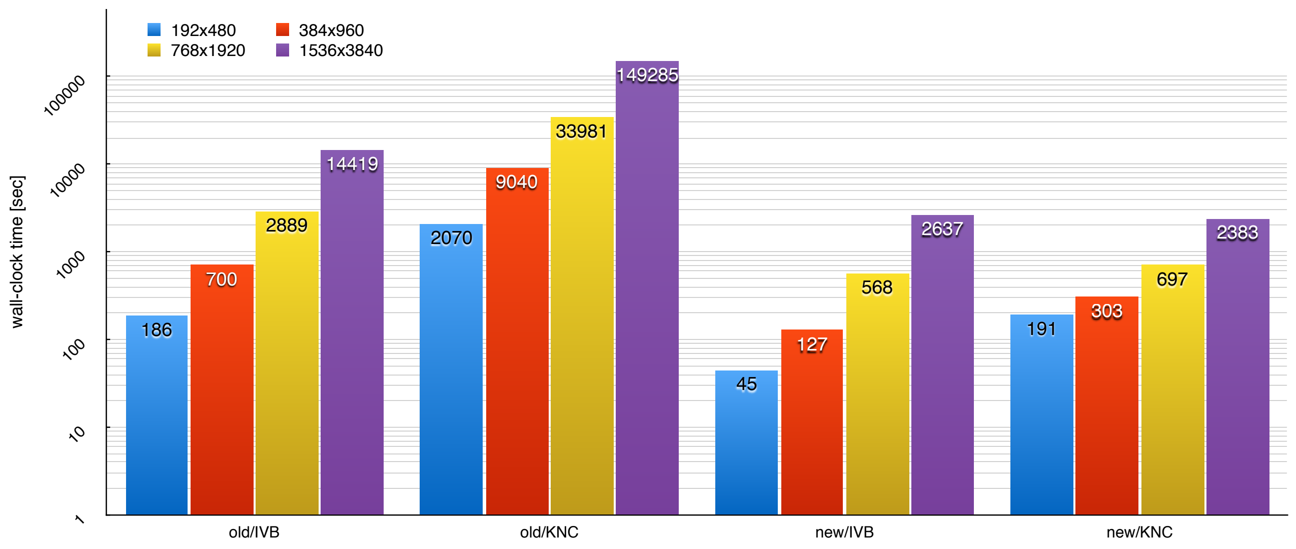


Fig. 5. Runtime in seconds of the original (old) and optimized (new) code on a compute node with dual-socket Sandy Bridge E5-2695 v2 CPU (IVB, total 24 cores/48 threads) and Xeon Phi 5100 (KNC, total 60 cores/240 threads). Different hardware configurations are shown on the horizontal axis, while the runtime in seconds is displayed on the vertical axis. Different problems sizes are colour-coded with the legend displayed on the top-left corner. For the reader's convenience, we also show the runtime for each problem size at the top of each corresponding bar.

automatically of benefit to Xeon as well. The inverse however is possible, if less than 2x speed-up is observed on XeonPhi, this means the code likely has further optimization potential.

We measured the application runtime on different problem sizes ranging from 192x480 to 1536x3840, which correspond to problem size ratio of 64. Few observations can be made from the figure. As expected, the application efficiency of small problem sizes (192x480, 384x960) on KNC is rather poor. In particular, we note that theoretically the runtime should differ by 4x for neighbouring problem sizes (the problem size increases by 4x in each step). In the case of IVB, both old and new versions of the application show about 4x increase in run-time with every problem-size step. This is not true for the new version running on KNC. Even for the largest problem sizes we see only 3.4x increase in runtime. We find it to be sufficiently close to 4x to argue that the overhead becomes somewhat irrelevant at such problem sizes.

We now focus on the largest problem size (1536x3840). There it can be seen that XeonPhi 5100 is ∼10% faster than dual-socket E5-2695 v2 CPU. As we earlier mentioned, theoretically XeonPhi 5100 is approximately twice as fast as our dual-socket IVB configuration. This implies that there is still a 2x speed-up potential for the application on KNC, which is consistent with our analysis at the end of Sect. 2.4. Unfortunately, due to limited amount of time allocated for this project and complexity of the application we were not able to act on our analysis, but it is our conviction that this will likely be able to recover the remaining ∼2x speed-up. We leave this for future work.

4. Conclusions

We conclude this white paper by claiming that XeonPhi is an excellent throughput optimized processor, and is capable of delivering outstanding performance. However, it takes serious amount of efforts, beyond what is being currently taught in textbooks and manuals, to tap into this performance. The biggest obstacle, in our opinion, is the immaturity of Intel Compiler Suites which prevents the user to exploit full richness of vector parallelism that XeonPhi offers. In this white paper we presented a possible solution to this but this forces the user to move beyond the boundaries of the Intel Compiler Suite. However, the benefit of all this is that the resulting code may offer performance portability across different processors, such as Xeon, XeonPhi and possibly GPU, on codes with a wide range of complexities that usually encountered in the field of High Performance Computing. It is unfortunate that the lack of mature industry accepted standard to explicitly use vector capabilities of modern processors in a portable manner creates major obstacles to developing applications that might be called legacy codes in a decade from now; but this will change.

Acknowledgements

EG would like to thank Alex Duran from Intel Corporation for his help with VTune, Intel Compiler Suite and low-level details of XeonPhi that made it possible to quickly identify, understand and fix various inefficiencies in the original code–efforts that ultimately led to the success of this project. This work was financially supported by the PRACE-1IP project funded in part by the EUs 7th Framework Programme (FP7/2007-2013) under grant agreement no. RI-261557. The work was achieved using the PRACE Research Infrastructure resources at SURFsara and Cineca.